LOG CENTRAL

eduardogamebooster@gmail.com Eduardo Bento

ericasugui@gmail.com Erica Suguimoto

ever@azul.dev Everton Jesus

fbzpanatto@hotmail.com Fabrizio Panatto

Projeto

Esta aplicação tem por objetivo centralizar todos os registros de erros de quaisquer outras aplicações, oferecendo consultas que permitam monitoramento e possível tratamento de acordo, por exemplo, com volume e gravidade das ocorrências.

index.js

api > src > services > mailer > index.js > ...

```
const nodemailer = require(`nodemailer`)
const { settings } = require('../../config')

class Mailer {
  constructor() {
    this.transport = nodemailer.createTransport({
      service: settings.mailer.service,
      auth: settings.mailer.auth
    })
  }

  send(to, subject, html) {
    return new Promise((resolve, reject) => {
      try {
        this.transport.sendMail({
          from: settings.mailer.from,
          to,
          subject,
          html
        }, error => {
          if (error) {
            throw error
          }

          resolve()
        })
      } catch (error) {
        reject(error)
      }
    })
  }
}

module.exports = new Mailer()
```

A construção da API entregue foi idealizada e realizada de maneira que um front-end para a aplicação deva tratar apenas da integração entre as ações do usuário e a própria API. Ou seja, os registros de erros devem ser entregues ao front-end de forma prática, eximindo da interface com o usuário a necessidade de realizar tarefas como, por exemplo, ordenação e paginação de informações.

Tecnologias

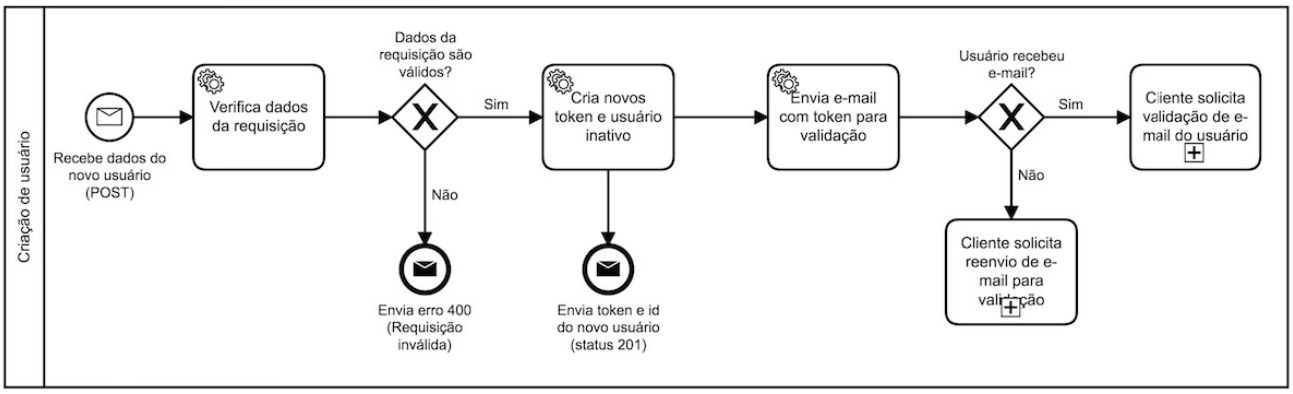
Aplicação

Autenticação

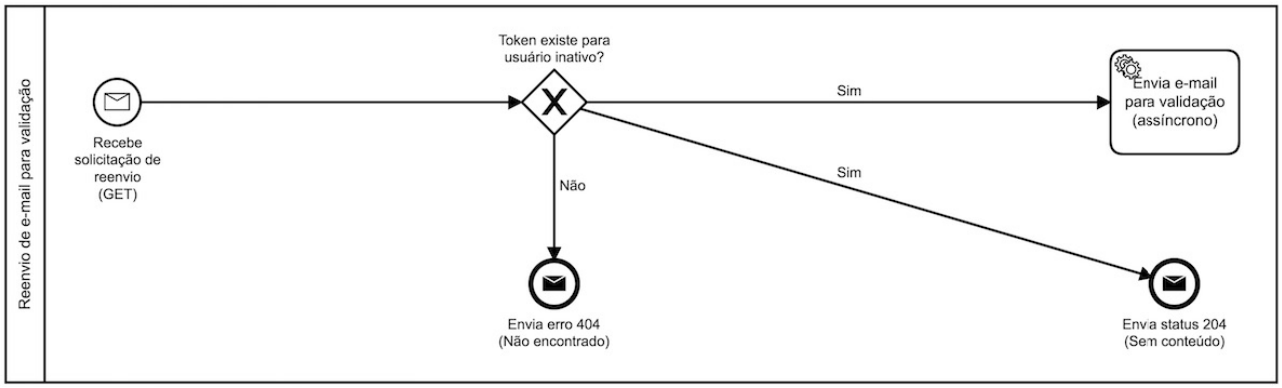
Pela natureza simples da aplicação proposta, o único conjunto de funcionalidades que contém regras de negócio mais complexas é o relacionado à autenticação, desde a criação de usuário até detalhes como validação do e-mail e recuperação de senha. Então, discorreremos sobre o processo, que é parte significativa da aplicação e faz uso de vários componentes nela implementados.

A seguir, fluxos resumidos para cada etapa, destacando o *endpoint* para acompanhamento e/ou testes via Swagger:

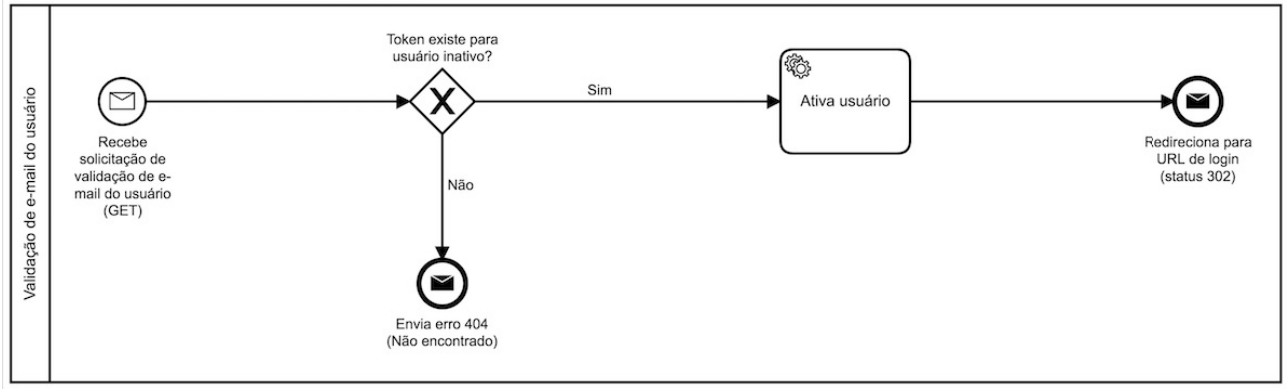
POST /User/sign-up



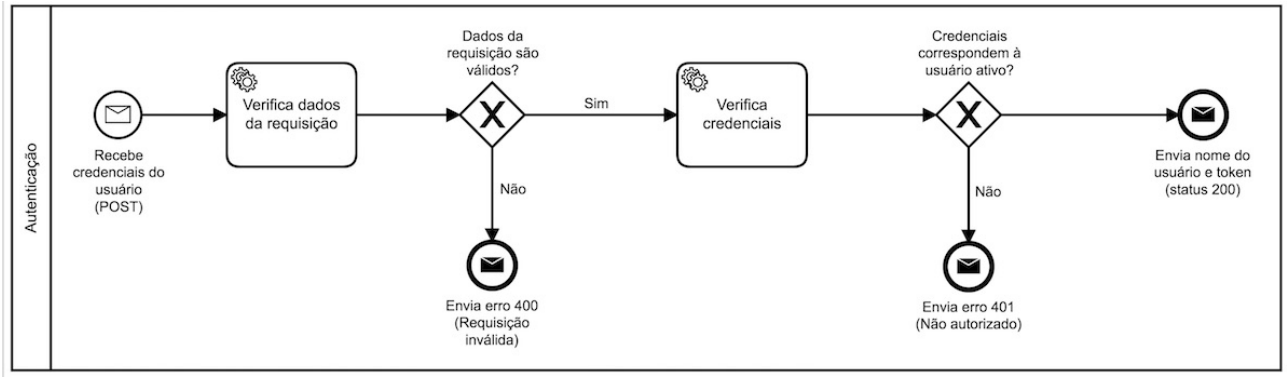
GET /User/sign-up/resend/{email}



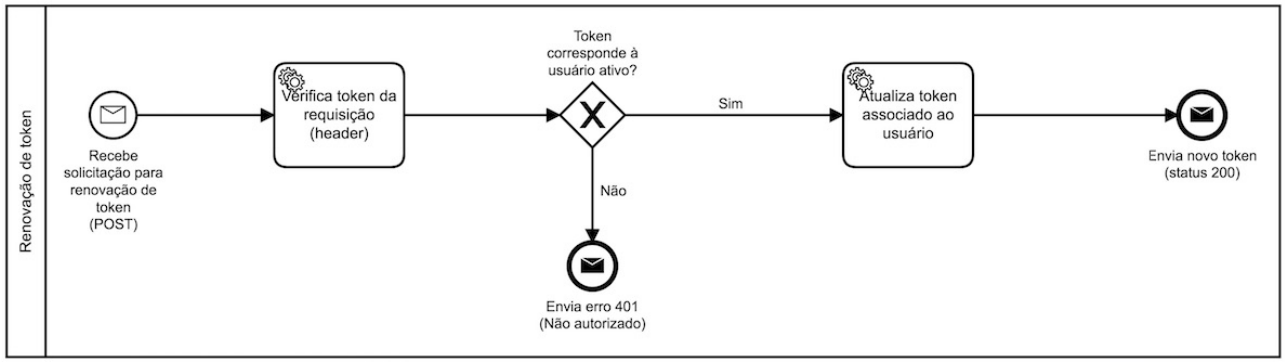
GET /User/sign-up/{token}



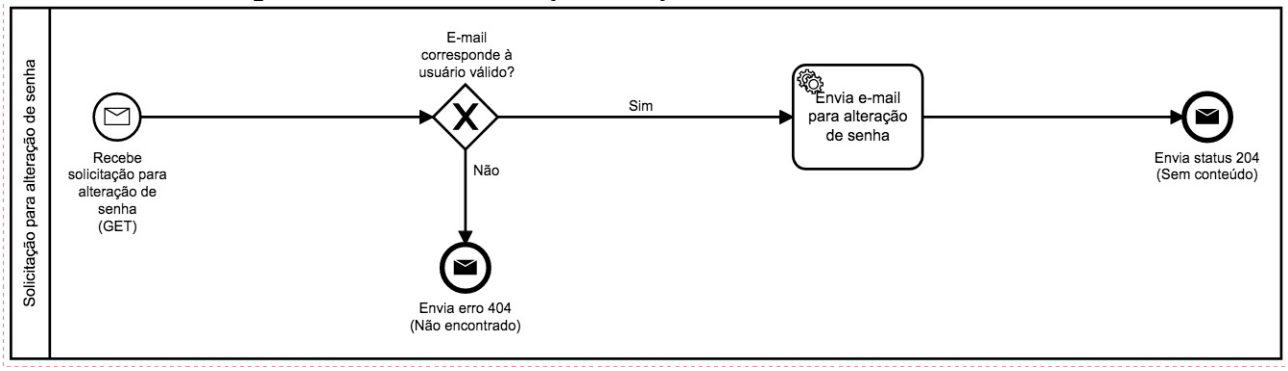
POST /User/sign-in



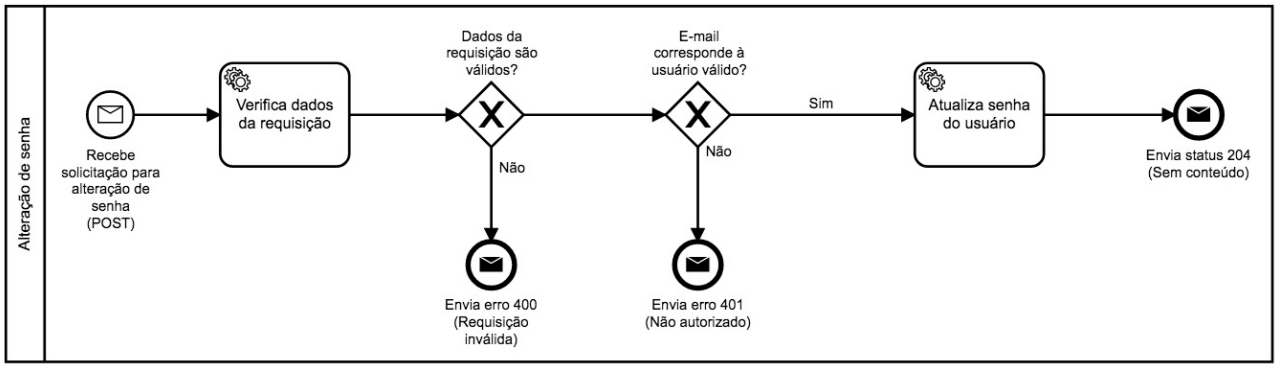
POST /User/token



GET /User/password-reset/{email}



POST /User/password



Logs

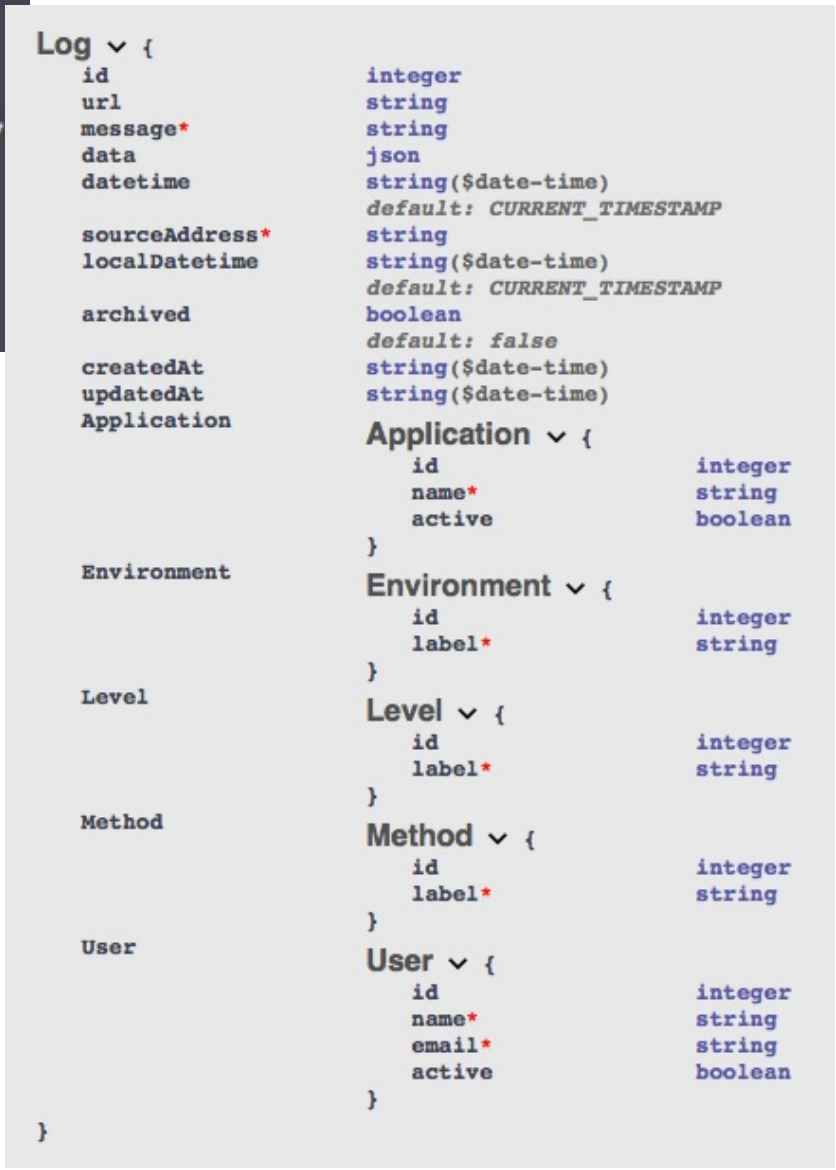
Também abordaremos a inserção e consulta dos logs, entidade central da aplicação, demonstrando os relacionamentos entre esta e as entidades de domínio criadas para assegurar a normalização da estrutura de dados.

Dentre as funcionalidades a demonstrar via Swagger, cabe destacar as funções de paginação, ordenação e filtros para a consultas destes *logs*, todas efetuadas a partir de *query parameters*.

```
{
  "application": {
    "id": 0
  },
  "environment": {
    "id": 0
  },
  "method": {
    "id": 0
  },
  "level": {
    "id": 0
  },
  "sourceAddress": "string",
  "datetime": 0,
  "url": "string",
  "message": "string",
  "data": "string"
}
```

Para inserção (via POST), o body enviado para o endpoint da entidade Log é composto conforme representado ao lado, indicando apenas a chave primária (campo id) de cada entidade associada.

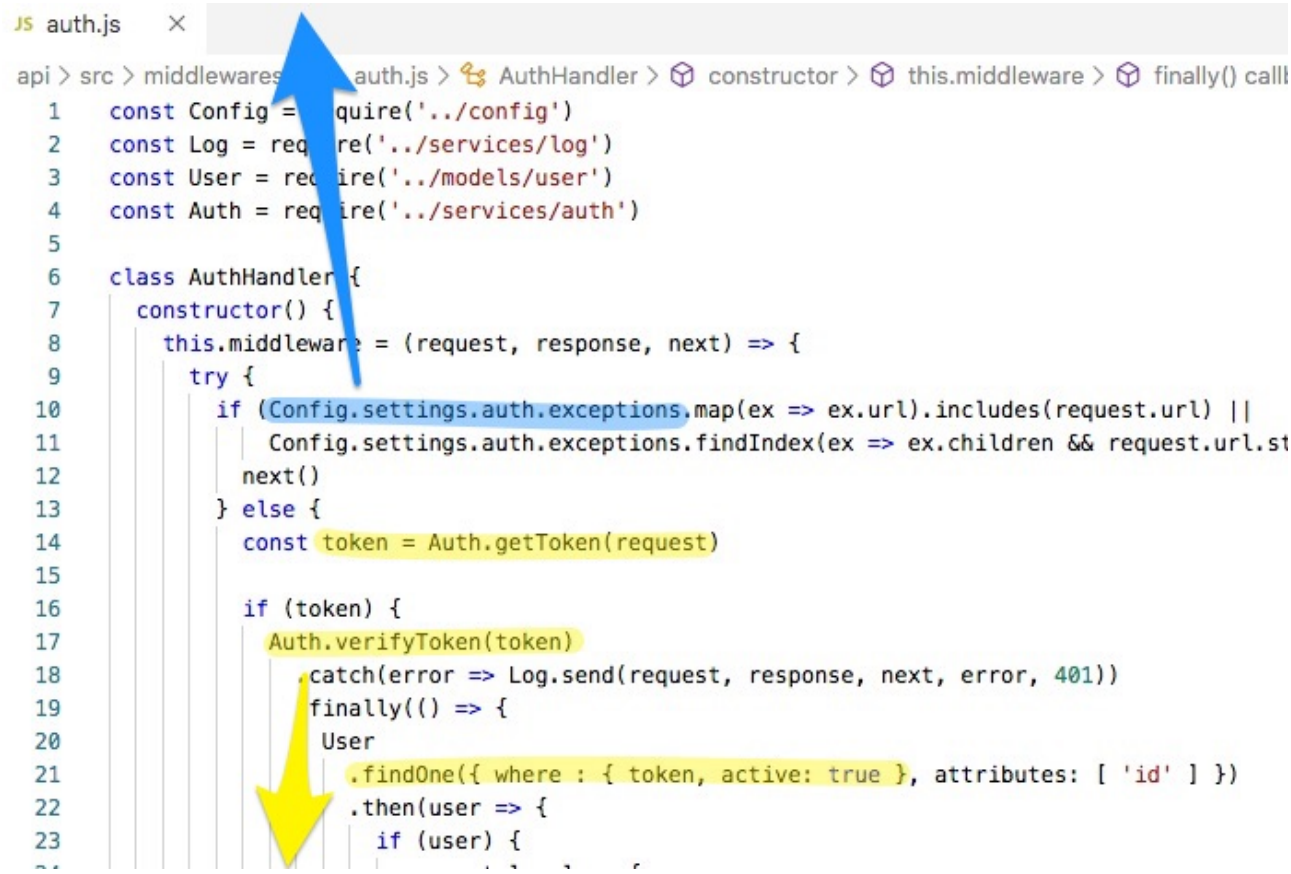
Já para as consultas (via GET), além da respectiva chave primária, cada entidade associada é retornada também contendo os campos exclusivos e/ou obrigatórios.



Middleware para autenticação

Todos os *endpoints* da aplicação são submetidos ao *middleware* exclusivo para autenticação, garantindo a segurança via login/e-mail e senha.

O acesso sem autenticação é permitido apenas para *endpoints* especificados como excessões no arquivo de configurações da aplicação.



Utilizando o padrão JWT, o gerenciamento de acesso é efetuado sem registro persistente da sessão do usuário no servidor.

Configurações

Um arquivo *JSON* concentra as principais configurações da aplicação, desde informações para a conexão ao banco de dados até o intervalo para expiração dos tokens.

As excessões para *endpoints*, citadas anteriormente, também são definidas neste arquivo.

Model

As entidades são definidas a partir de uma classe modelo/genérica que implementa métodos comumente utilizados em operações básicas.

model.js

api > src > utils > model.js > ...

```
1   const Sequelize = require('sequelize')
2
3   class Model extends Sequelize.Model {
4     static findById(id, attributes = undefined, include = undefined) {
5       return this.findOne({
6         where: { id },
7         attributes,
8         include
9       })
10    }
11
12    static updateById(id, body) {
13      return this.update(body, {
14        where: { id }
15      })
16    }
17
18    static destroyById(id) {
19      return this.destroy({
20        where: { id }
21      })
22    }
23  }
24
25  module.exports = Model
```

log.js

api > src > models > log.js > Log > init

```
1   const { DataTypes } = require('sequelize')
2   const Model = require('../utils/model')
3
4   class Log extends Model {
5     static init(sequelize) {
6       return super.init({
7         url: DataTypes.STRING,
8         message: {
9           type: DataTypes.STRING,
10          allowNull: false,
11          validate: {
12            notEmpty: true
13          }
14        },
15        data: DataTypes.JSON,
16        datetime: {
17          type: DataTypes.DATE,
18          allowNull: false,
19          defaultValue: DataTypes.NOW
20        },
21        sourceAddress: {
```

Assim, cada entidade iniciada pode usufruir de tais métodos por herança, evitando a repetição de códigos e reduzindo o tempo despendido para execução de implementações e até mesmo de testes.

Service

Para evitar a necessidade de implementação de métodos comuns para cada uma das entidades representadas neste formato descrito no item anterior, optamos pela implementação de uma classe genérica para a camada de serviço.

Para entidades cujas regras de negócio necessitem apenas de operações *CRUD* padrão, o service será criado/estendido em tempo de execução.

service.js

api > src > utils > service.js > ...

```
1   const Log = require('../services/log')
2   const models = require('../models')
3
4   class Service {
5     constructor(modelFile, attributes = undefined, include = undefined) {
6       this.model = require('../models/' + modelFile)
7       this.attributes = attributes
8       this.include = include
9     }
10
11    getById(request, response, next) {
12      try {
13        this.model.findById(request.params.id, this.attributes, this.include)
14          .then(data => {
15            if (data) {
16              response.json(data)
17            } else {
18              Log.send(request, response, next, null, 404)
19            }
20          })
21          .catch(error => Log.send(request, response, next, error))
22      } catch (eror) {
23        Log.send(request, response, next, error)
24      }
25    }
26
27    create(request, response, next) {
28      try {
29        this.model.create(this.bodyFormat(request.body))
30          .then(newData => {
31            const attributes = !this.include ? [ ...Object.keys(newData._changed),
32
33            this.model.findById(newData.id, attributes, this.include)
34              .then(data => response.status(201).json(data))
35              .catch(error => Log.send(request, response, next, error))
36          })
```

Esta camada será responsável pelas diversas integrações entre os *controllers* (onde serão definidas as rotas) e o banco de dados — sempre através do *ORM Sequelize*.

A depender das regras de negócio, dada classe de serviço poderá herdar os métodos padrões da classe básica, conter novos métodos e/ou até mesmo subscrever métodos existentes.

Controller

Também para dispensar a necessidade de reimplementação dos mesmos métodos associados à cada entidade, um *controller* básico contém as rotas mais comuns.

log-service.js

api > src > services > log > log-service.js > Config

```
const Log = require('.')
const Service = require('../../utils/service')
const models = require('../../models')
const Config = require('../../config')

class LogService extends Service {
  archive(request, response, next) {
    this.model
      .updateById(request.params.id, { archived: true })
      .then(data => {
        if (data[0] > 0) {
          response.status(204).send()
        } else {
          Log.send(request, response, next, null, 404)
        }
      })
      .catch(error => Log.send(request, response, next, error, 404))
  }

  create(request, response, next) {
    models.User
      .findOne({ where: { token: request.headers[Config.settings.auth.key] }, attributes: [ 'id' ] })
      .then(user => {
        if (user) {
          request.body.user = { id: user.id }

          super.create(request, response, next)
        } else {
          Log.send(request, response, next, null, 401)
        }
      })
      .catch(error => Log.send(request, response, next, error))
  }
}

module.exports = new LogService('log', undefined, include)
```

controller.js

api > src > utils > controller.js > ...

```
const express = require('express')
const Service = require('./service')

class Controller {
  constructor(serviceOrModelFile) {
    this.router = express.Router()
    this.service = typeof serviceOrModelFile === 'object' ? serviceOrModelFile : new Service(serviceOrModelFile)
  }

  setDefaultRoutes() {
    this.router.get('/', (request, response, next) => this.service.getAll(request, response, next))

    this.router.get('/:id', (request, response, next) => this.service.getById(request, response, next))

    this.router.post('/',
                     (request, response, next) => this.service.validator(request, response, next),
                     (request, response, next) => this.service.create(request, response, next))

    this.router.patch('/:id',
                      (request, response, next) => this.service.validator(request, response, next),
                      (request, response, next) => this.service.update(request, response, next))

    this.router.delete('/:id', (request, response, next) => this.service.delete(request, response, next))
  }
}

module.exports = Controller
```

Para entidades cujas regras de negócio necessitem apenas de operações CRUD padrão, o *controller* será criado/estendido em tempo de execução.

Já para entidades que necessitem de mais operações, o código implementado estende o *controller* padrão e implementa apenas as demais rotas.

Conforme exemplo acima, rotas padrões indesejadas são evitadas ao atribuir um retorno padrão de status 405 (não permitido).

Dificuldades

Tempo escasso dos integrantes

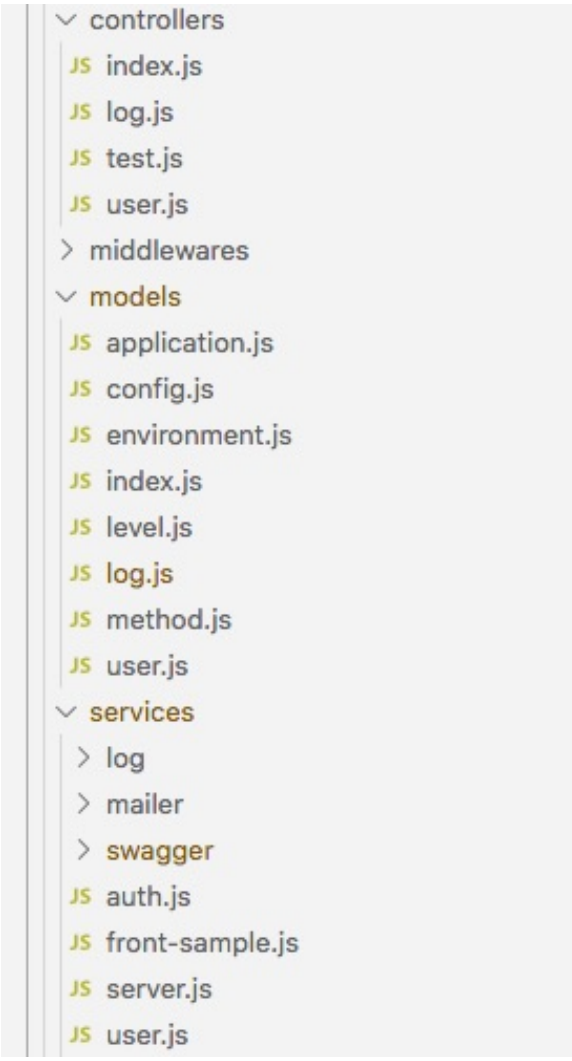
Devido à ausência de tempo, ocasionada por mudanças de rumo de carreiras e vida pessoal de alguns integrantes — incluindo a desistência de uma deles, ainda que por motivos extremamente justificáveis —, tivemos dificuldade em nos organizar para cumprir todos os objetivos.

Como a divisão das tarefas tornou-se um ponto crítico a medida que as agendas profissionais de alguns membros foi ficando reduzida, fomos obrigados a deixar de lado uma das funcionalidades que, embora não explicitada no requisito, julgávamos ser um diferencial para o projeto.

Mudança de programação

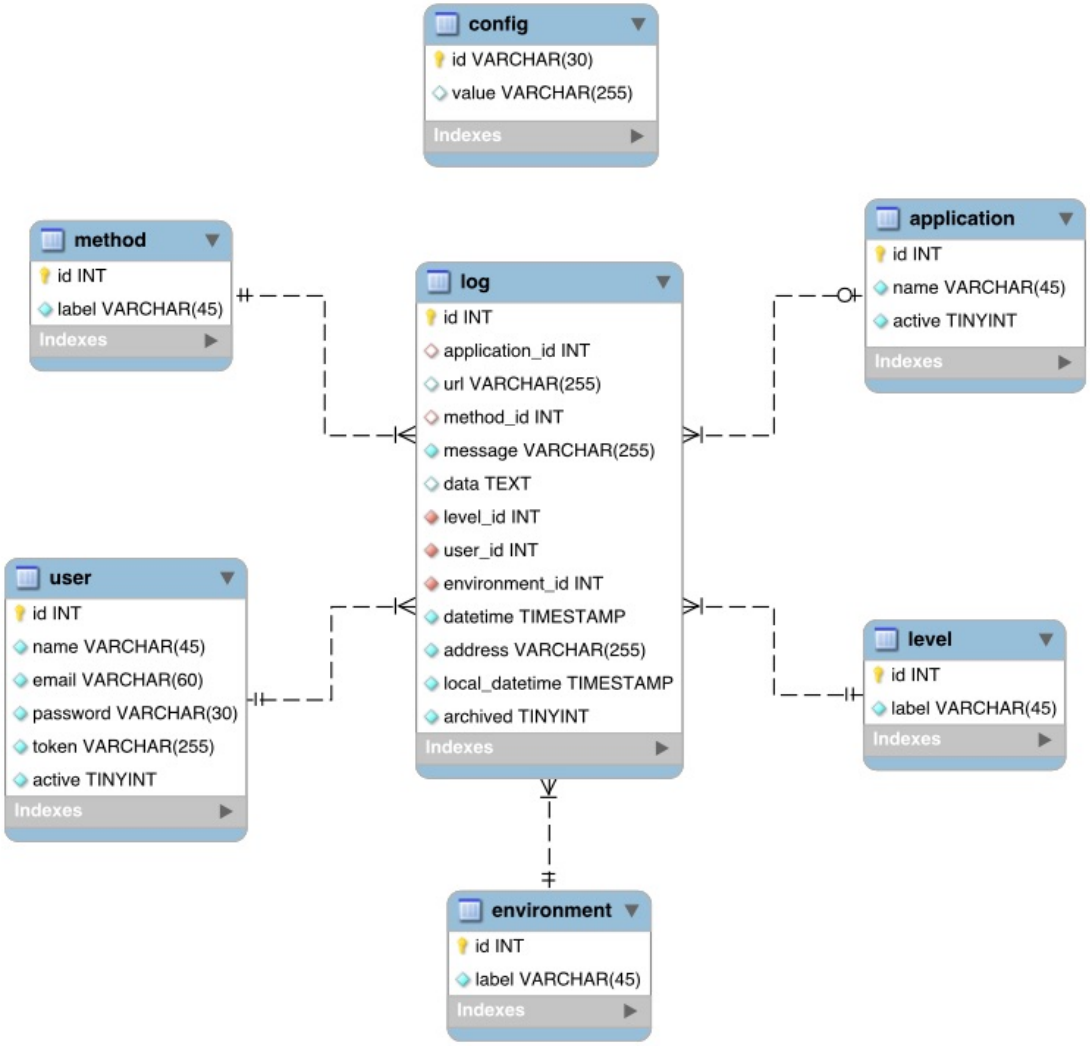
Duas aulas presenciais que seriam exclusivamente dedicadas ao projeto final foram, por motivos de força maior, convertidas para o formato online, então com conteúdo aplicado.

Ficou comprometido o principal alicerce de nosso planejamento, que seriam dois *sprints* presenciais para codificação da aplicação já idealizada.



Assertividade

Acreditamos que produzir o modelo de dados e a documentação — utilizando *Swagger* — antes de iniciar a codificação das funcionalidades tenha sido nosso maior acerto.



Tais definições foram essenciais para reduzir a incidência de dúvidas sobre o que seria necessário implementar para cada item listado no escopo — principalmente devido à necessidade de identificar requisitos pouco detalhados, a partir de *screenshots* acrescentados ao resumo do projeto.

Após identificadas as necessidades, a disposição de todos para discutir as possíveis soluções durante pequenos intervalos das aulas presenciais também foi essencial para o resultado apresentado.

https://api.logcentral.azul.dev