API NODE

PADRONIZAÇÃO

Julho de 2021

HISTÓRICO DE ALTERAÇÕES

ALTERAÇÃO	RESPONSÁVEL	DATA
Adição padrões API-Node	Felipe Libório, José Pirangaba	14/07/2021

Versão atual: 1.0.

1. APRESENTAÇÃO

Este documento apresenta os padrões que devem ser utilizados nas nomenclaturas e localização dos arquivos, geração de código, consultas SQL, *services*, *controllers*, estrutura dos módulos, parâmetros e respostas de requisições da arquitetura utilizada no projeto API NODE.

2. OBSERVAÇÕES GERAIS

2.1. OBSERVAÇÕES GERAIS - CÓDIGO

- Variáveis e funções devem ser nomeadas em *camelCase*;
- Classes devem ser nomeadas em *PascalCase*;
- Constantes *"hard coded"* devem ser nomeadas em CAIXA_ALTA;
- O código deve ser escrito em português. Incluindo nomes de variáveis, classes e métodos.
- Código passível de erros não tratados será rejeitado (e.g: tentar acessar usuario.nome quando usuário pode ser indefinido, ao invés de acessar usuario?.nome ou validar esse acesso de alguma maneira).
- A biblioteca *day.js* deverá ser utilizada para a manipulação de datas quando não for conveniente utilizar apenas bibliotecas padrão do Javascript. Alternativas como *moment.js* não devem ser utilizadas.

2.2. CAPTURAR DATA/HORA ATUAL

Ao se recuperar a data/hora atual do servidor com a biblioteca day.js, deve-se adicionar adicionar ".tz()" a chamada para que a configuração de fuso horário definida para o projeto seja utilizada. Como no exemplo a seguir:

```
let dadosUsuario = await dadosMemcached.getDados(req.headers['x-access-token']);
if (!dadosUsuario) throw { status: 0, mensagem: 'Erro ao acessar dados de login'};
let dataConsulta = dayjs().tz().format("YYYY-MM-DD HH:mm:ss");
let placas = dadosUsuario.placas;
let local = req.query.local;
```

2.3. NOMENCLATURA E LOCALIZAÇÃO DOS ARQUIVOS

Os nomes de *controller* e services devem usar *PascalCase*, no código e em demais arquivos deve ser usado o *camelCase*;

Controllers devem ser criados nos diretórios correspondentes: "frotas", "safety", "shownet" ou "relatórios";

Cada relatório deverá ter um controller próprio no diretório adequado, e.g. para um relatório de viagens gerado por um usuário do Gestor será ".../src/controllers/relatorios/**frotas**/Viagens.js", para um relatório destinado a uso interno será ".../src/controllers/relatorios/**shownet**/DiasRastreados.js";

Os nomes de arquivos deverão estar em português.

2.4. CONSULTAS SQL

As consultas aos bancos de dados deverão ser criadas em ***Services*** e consultas SQL deverão utilizar o Knex como *query builder*. ***Raw queries*** precisarão ser justificadas à homologação e ao responsável pela API.

Consultas a dados do usuário feitas por *endpoints* acessíveis a usuários do Gestor deverão se atentar a se o usuário logado possui ou não acesso aos dados retornados. **Os dados de identificação do usuário e respectivo cliente DEVEM ser recuperados do *CACHE*** sempre que disponíveis utilizando-se o token enviado na requisição ao endpoint e o *helper* de acesso ao *cache*.

Consultas muito demoradas que façam *join* em várias tabelas deverão ser subdivididas em consultas menores para que as tabelas envolvidas não fiquem travadas durante muito tempo. Toda consulta estará sujeita a aprovação pelo setor de homologação.

O diretório *models* é depreciado e nenhum *model* deverá ser criado nele.

2.5. INSTALAÇÃO DE MÓDULOS

Caso os módulos já adicionados ao projeto não atendam o desenvolvimento de uma nova funcionalidade e um novo módulo precise ser adicionado ao projeto, essa adição deverá ser feita utilizando-se especificamente do NPM e as alterações nos arquivos *package.json* e *package-lock.json* resultantes dessa adição deverão ser incluídas no *pull-request* da atividade.

Os módulos novos precisarão ser compatíveis com o *Elastic Beanstalk* essa validação poderá ser feita pela equipe de homologação.

2.6. VERSÃO DA NODE UTILIZADA

A versão do Node.js utilizada por este projeto atualmente é a 14.16.0. Qualquer atualização na versão do Node.js será feita exclusivamente pelo responsável pelo projeto e é recomendável que os desenvolvedores utilizem esta versão ao trabalhar no projeto para evitar qualquer incompatibilidade no código.

3. *SERVICES*

Todas as funções ou classes exportadas deverão validar seus parâmetros e lançar um erro adequado caso os argumentos recebidos sejam inválidos. A fim de padronizar as respostas de erro retornadas ao usuário o padrão a seguir deverá ser adotado.

Os erros deverão ser lançados em um proto objeto no seguinte formato:

{

status,

code,

mensagem,

err

}

Em que **“*status*”** é numérico e para erros deverá ser negativo. **“*code*”** é numérico e é um dos códigos de resposta HTTP. **“mensagem”** é uma descrição em português do erro. E **“err”** é o objeto do erro completo ou parcial que poderá ser retornado quando conveniente ao usuário. *“status”*, *“code”* e “mensagem” sempre deverão estar no erro lançado.

Exemplo:

```
async function getMacros(colunas = '*') {
  return new Promise((resolve, reject) => {
    if (colunas != '*' && (!Array.isArray(colunas) || !colunas.length)) {
      return reject({status: -1, code: 400, mensagem: 'Macros Omnilink: colunas inválido, esperando array'});
    }

    db.omnilink('macro')
    .select(colunas)
    .then(res => {
      resolve(res)
    })
    .catch(err => {
      if (err.code == 'ER_BAD_FIELD_ERROR') {
        reject({status: -1, code: 400, message: 'Macros Omnilink: coluna inválida informada', err: { sqlMessage: err.sqlMessage }});
      } else {
        reject({ status: -99, code: 500, message: 'Macros Omnilink: erro ao consultar banco de dados ', err: { sqlMessage: err.sqlMessage, message: err.message } });
      }
    });
  });
}
```

4. *CONTROLLERS*

Todas as funções em que uma requisição é respondida deverão ter seu corpo dentro de um ***try…catch*** e retornar um erro no mesmo formato que o lançado pelos *services*.

Todos os parâmetros de requisição deverão ser validados, aqueles que não forem validados por um *service* precisarão ser validados no *controller* da requisição.

A validação dos parâmetros deverá lançar uma exceção no mesmo formato descrito para as exceções lançadas pelos *services*.

As validações de parâmetros deverão usar os *helpers* de validação quando eles atenderem o tipo de validação que se deseja fazer.

Novas validações poderão ser adicionadas ao *helper* de validação mas estarão sujeitas a validação pela equipe de homologação e pelo responsável pela API.

As funções de requisição deverão retornar um código de *status* HTTP apropriado em caso de erro. Códigos 5XX estão restritos a erros desconhecidos ou não relacionados à entrada do usuário.

Todas as respostas do tipo json deverão conter um *status* numérico, esse status será negativo em caso de falha e não negativo em caso de sucesso. “1” será o *status* de sucesso normalmente utilizado.

Exemplo de função que trata uma requisição externa em um controller:

```
176
177
178  async function validarUsuarioTelefoneAdm(req, res) {
179    try {
180      const { chaveCliente, tipoChave, telefoneUsuario } = req.query;
181      const resposta = await Cliente.validarUsuarioPorTelefone(chaveCliente, tipoChave, telefoneUsuario);
182
183      return res.json(resposta);
184    } catch(err) {
185      let resposta = err;
186
187      if (err.status == undefined) {
188        console.log('[controllers][ClienteController][validarUsuarioAdm]', err || '');
189        resposta = { status: 0, code: 500, mensagem: "erro", err };
190      }
191
192      return res.status(resposta.code || 500).json(resposta);
193    }
194  }
195
```

5. ESTRUTURA DOS MÓDULOS

Todos os módulos (exceto classes) deverão seguir o seguinte padrão de organização:

Exemplo:

```
const DM = require('../../../helpers/dadosMemcached');
const Formatacao = require('../../../helpers/formatacao');
const Validacao = require('../../../helpers/validacao');
const Relatorios = require('../../../helpers/relatorios');


module.exports = {
  postRelatorio,
  gerarRelatorio
}

const SLICE_PLACAS = 25;
const TIMEOUT = 45000;

const DIRECOES = [
  { pt: 'Norte', en: 'North', es: 'Norte' },
  { pt: 'Nordeste', en: 'Northeast', es: 'Noreste' },
  { pt: 'Leste', en: 'East', es: 'Este' },
  { pt: 'Sudeste', en: 'Southeast', es: 'Sureste' },
  { pt: 'Sul', en: 'South', es: 'Sur' },
  { pt: 'Sudoeste', en: 'Southwest', es: 'Sur oeste' },
  { pt: 'Oeste', en: 'West', es: 'Oeste' },
  { pt: 'Noroeste', en: 'Northwest', es: 'Noroeste' },
];

const MAPA_STATUS = {···
};

const TRADUCOES = {···
}

async function postRelatorio(req, res) {
  try {
    let { inicio, fim, placas, parte } = req.body;

    Validacao.placas(placas);
    Validacao.periodo(inicio, fim);

    const partes = Math.ceil(placas.length / SLICE_PLACAS);
    parte = Number(parte);
    if (!Number.isInteger(parte) || parte < 1 || parte > partes) {
      throw { status: 1, code: 400, mensagem: 'Parte inválida, esperando inteiro entre 1 e ' + partes };
```

6. RELATÓRIOS

Os relatórios em geral serão chamados por uma rota POST, que deverá ser autenticada com o token de usuário ou o token do Shownet.

As rotas dos endpoints de relatórios deverão explicitar que a chamada trata-se de um relatório e a que projeto esse relatório pertence, seguindo o seguinte padrão: "…/relatorios/produto/nome", e.g. "…/relatorios/frotas/viagens".

No arquivo de rotas, os relatórios deverão estar todos agrupados no prefixo destinado a eles:

```
215
216  /** ROTAS RELATÓRIOS */
217  routes.prefix('/api/relatorios/', (relatorios) => {
218    /** GESTOR */
219
220    relatorios.route('/mensagensLivres').post(Auth.verifyAuth, MensagemLivre.getRelatorioMensagensLivres);
221    relatorios.route('/mensagensFormatadas').post(Auth.verifyAuth, MensagemFormatada.getRelatorioMensagensFormatadas);
222    relatorios.route('/alarmesBloqueios').post(Auth.verifyAuth, AlarmesBloqueios.getRelatorioAlarmesBloqueios);
223    relatorios.route('/teleeventos').post(Auth.verifyAuth, Teleeventos.getRelatorioTeleeventos);
224    relatorios.route('/temperaturaSaver').post(Auth.verifyAuth, TemperaturaSaver.getRelatorioTemperaturaSaver);
225
226    relatorios.route('/posicoes').post(Auth.verifyAuth, Posicoes.getRelatorioPosicoes);
227    relatorios.route('/historicoPosicaoIscas').post(Auth.verifyAuth, HistoricoIsca.getHistoricoPosicaoIscas);
228    relatorios.route('/pontoInteresseBasico').post(Auth.verifyAuth, PontoInteresse.getPontoInteresseBasico);
229    relatorios.route('/mensagensRecebidas').post(Auth.verifyAuth, MensagensRecebidas.getRelatorioMensagensRecebidas);
230    relatorios.route('/eventosOcorridos').post(Auth.verifyAuth, EventosOcorridos.getRelatorioEventosOcorridos);
231
232    relatorios.route('/pontoInteresseTnz').post(Auth.verifyAuth, PontoInteresseTnz.getPontoInteresseTnz);
233    relatorios.route('/getAreaEventos').post(Auth.verifyAuth, AreaEventos.getAreaEventos);
234    relatorios.post('/eventosMonitorado', Auth.verifyAuth, Tornozeleiras.eventosMonitoradoReq);
235    relatorios.route('/detalhamentoMotorista').post(Auth.verifyAuth, DetalhamentoMotorista.getDetalhamentoMotoristaReq);
236    relatorios.post('/movimentacaoIndevida', Auth.verifyAuth, MovimentacaoIndevida.getMovimentos);
237    relatorios.post('/violacaoBateria', Auth.verifyAuth, ViolacaoBateria.getViolacoes);
238    relatorios.post('/excessoVelocidade', Auth.verifyAuth, ExcessoVelocidade.postRelatorio);
239    relatorios.post('/kmRodado', Auth.verifyAuth, KmRodado.getKmRodadosReq);
240    relatorios.post('/tacografo', Auth.verifyAuth, Tacograph.postTacografo);
241    relatorios.post('/situacaoVeiculo', Auth.verifyAuth, SituacaoVeiculo.getSituacaoVeiculoReq);
242    relatorios.post('/hodometro', Auth.verifyAuth, Hodometro.postRelatorioHodometro);
243    relatorios.post('/deslocamento', Auth.verifyAuth, DeslocamentoAcumulado.postRelDeslocamentoAcumulado);
244    relatorios.post('/ociosidade', Auth.verifyAuth, Ociosidade.postGetOciosidade);
245    relatorios.post('/resumoJornada', Auth.verifyAuth, ResumoJornada.getTrack);
246    relatorios.post('/diasRastreados', Auth.verifyAuthShownet, DiasRastreados.postDiasRastreados);
247    relatorios.post('/analiseTempo', Auth.verifyAuth, analiseTempo.postAnaliseTempo);
248
```

6.1. PARÂMETROS

Normalmente, os relatórios receberão como parâmetros um objeto com as propriedades "placas" (ou "motoristas"), "inicio", "fim" e "parte" assim grafadas.

"placas" será sempre um array de string contendo as placas usadas como chave no relatório. E **todas** as placas deverão ser informadas nesta propriedade. Ou seja, independentemente de o Gestor permitir que o usuário gere o relatório por grupos ou que selecione todos os veículos, isso terá de ser tratado do lado do Gestor para que seja enviado um array de placas à API a fim de simplificar a chamada e seu tratamento.

De forma análoga às placas, os **"motoristas"** deverão ser enviados em um array de strings. Os motoristas deverão ser identificados pelo seu chaveiro serial a menos que haja uma justificativa para o uso de outra chave para a sua identificação, e.g. o relatório incluirá motoristas que não possuem chaveiro serial.

“inicio” e **“fim”** deverão ser datas no formato “YYYY-MM-DD HH:mm:ss” independentemente do formato delas no gestor. Por exemplo, se um relatório é diário e não se escolhe a hora, “inicio” e “fim” ainda deverão ser passados como “YYYY-MM-DD 00:00:00” e “YYYY-MM-DD 23:59:59” respectivamente. A função **“periodo”** do helper de validação deverá ser usada para validar os parâmetros “inicio” e “fim” e o intervalo máximo permitido para relatórios de usuários do Gestor deve ser 31 dias.

“parte” deve ser um inteiro representando a parte atual do relatório, começando em 1. Essa propriedade aplica-se apenas a relatórios particionados.

Parâmetros adicionais deverão ser nomeados em português e em camelCase e seus nomes deverão ser descritivos e não abreviados (e.g. “horaInicio” e não “hIni”).

Dados que possam ser obtidos pelo memcached não devem ser passados como parâmetro, e.g. não há porque passar o id do usuário em um relatório do Gestor.

6.2. RESPOSTA

A resposta dos relatórios deverá ser do tipo json e o objeto retornado terá as propriedades “status”, “parte”, “partes”, “chaveAuditoria” e “dados” em caso de sucesso, em caso de erro o formato da resposta será o descrito na seção *Services* deste documento.

“status” deverá ser um número inteiro positivo para respostas de sucesso, geralmente “1”.

“parte” indica a parte atual do relatório. É aplicável apenas a relatórios particionados e deve ser igual a parte recebida na chamada do endpoint.

“partes” é um inteiro maior ou igual a um e representa o número de partes existentes em um relatório particionado.

“chaveAuditoria” é a chave gerada pela função auditoria exportada pelo helper de relatórios. Se o relatório for particionado, cada parte deverá gravar uma entrada de auditoria própria e retornar a chave de auditoria na resposta.

“dados” conterá o relatório e demais dados que ele possa retornar, por exemplo uma lista de falhas. Se “dados” for retornar algo além do próprio relatório, deverá ser um objeto e o relatório deverá estar contido em uma propriedade chamada “relatorio” dentro do objeto dados.

As respostas dos relatórios deverão estar localizadas de acordo com o idioma salvo no Memcached. Isso se aplica aos relatórios cuja chamada faz uso do token de usuário do Gestor como método de autorização. Há helpers dedicados à formatação e localização de diferentes tipos de dados.

6.3. RELATÓRIOS PARTICIONADOS

Relatórios que consultem a tabela “historico” do banco Dynamo (e.g. todas as consultas disponíveis no service HistoricoVeiculo) deverão ser subdivididos em várias chamadas. Essa subdivisão será feita utilizando-se a chave usada na consulta a tabela “historico” (placa ou chaveiro serial do motorista).

Esse particionamento deverá ser realizado na função responsável por receber e responder a chamada do relatório. Cada parte deverá estar limitada a no máximo 100 placas ou 25 motoristas independentemente do intervalo do relatório.

Todas as requisições de um relatório devem ter os mesmos parâmetros, exceto pela propriedade “parte”. Ou seja, se um relatório está sendo gerado para 10000 placas em todas as chamadas desse relatório as 10000 placas deverão ser enviadas no array “placas” e a fatia correspondente a parte atual será utilizada na geração do relatório.

Exemplo:

```
22
23  async function postRelatorio(req, res) {
24    try {
25      let { inicio, fim, motoristas, parte } = req.body;
26      let { idioma, id_cliente: idCliente } = await DM.getDados(req.headers['x-access-token']);
27
28      await Validacao.motoristas(motoristas, idCliente);
29      Validacao.periodo(inicio, fim);
30
31      const partes = Math.ceil(motoristas.length / SLICE_MOTORISTAS);
32      parte = Number(parte);
33      if (!Number.isInteger(parte) || parte < 1 || parte > partes) {
34        throw { status: -1, code: 400, mensagem: 'Parte inválida, esperando inteiro entre 1 e '+partes };
35      }
36
37      motoristas = motoristas.slice((parte - 1) * SLICE_MOTORISTAS, parte * SLICE_MOTORISTAS);
38
39      const dados = await gerarRelatorio(motoristas, inicio, fim, idioma, TIMEOUT);
40
41      let chaveAuditoria = Relatorios.auditoria({
42        relatorio: "Resumo Diário dos Motoristas",
43        inicio,
44        fim,
45        chaves: motoristas
46      }, req.headers['x-access-token']);
47
48      return res.status(200).json({ status: 1, parte, partes, chaveAuditoria, dados });
49    } catch (err) {
50      let resposta = err;
51      if (err.status == undefined) {
52        console.log('[controllers][relatorios][frotas][ResumoDiarioMotoristas][postRelatorio]', err || '');
53        resposta = { status: 0, code: 500, mensagem: "erro", err };
54      }
55
56      return res.status(resposta.code || 500).json(resposta);
57    }
58  }
59
```