
# WSuite - Mapeamento dos endpoints da API

**Por W Technology**

**São Paulo SP**

**2022**

**Versão 1.0**

# SUMÁRIO

## 1. Rotas de autenticação

O conjunto de rotas denominadas para autenticação referem-se a todos os métodos utilizados para autenticar o usuário para a utilização do sistema, com métodos que serão chamados em todas as requisições feitas pela parte de interface gráfica, ou seja, o front-end. No desenvolvimento de software, sites ou aplicativos, segue-se um padrão de divisão da aplicação a ser criada em dois campos: back-end e front-end. O front-end é responsável pela conexão da implementação do sistema (tudo que o usuário não pode ver, como banco de dados, servidores, segurança, ou seja, tudo que o back-end normalmente executa) com o usuário, pensando na melhor usabilidade do cliente final.

As rotas de autenticação irão trabalhar com as tecnologias:

- Bcrypt

  Processo de criptografia baseado em algoritmo hash, desenvolvido em 1999 por Niels Provos e David Mazières. O objeto do Bcrypt é esconder senhas inseridas por usuário em uma aplicação em forma de texto "normal" para texto indecifráveis. Processo muito usado em aplicações onde é guardado senhas no banco de dados.
- JWT

  Técnica de autenticação remota entre dois elementos. O Json Web Token, definido na RFC 7519, é um dos modos mais utilizados para autenticar usuários que desejam consumir APIs baseadas no padrão REST.
- Passport (1)

  Passport é um middleware de autenticação. O passport funciona utilizando o conceito de estratégias, onde elas serão usadas para autenticar solicitações. Existem diversas formas de estratégias, fica a cargo do desenvolvedor a criação ou utilização de alguma.
- Strategy

  Estratégia utilizada no Passport. secretOrKey é uma string ou buffer que contém a chave pública secreta (simétrica) ou codificada em PEM (assimétrica) para verificar a assinatura do token. OBRIGATÓRIO a menos que secretOrKeyProvider seja fornecido. secretOrKeyProvider é um retorno de chamada no formato function secretOrKeyProvider(request, rawJwtToken, done), que deve chamar “done” com uma chave pública secreta ou codificada em PEM (assimétrica) para a combinação de chave e solicitação fornecida. done aceita argumentos no formato function done(err, secret). Observe que cabe ao implementador decodificar “rawJwtToken”.

### 1.1. Entrar no sistema

Para entrar no sistema, o usuário deverá chamar a rota de “signin”, por meio do caminho URL “/v1/signin”. Para esta rota, a requisição deve vir acompanhada de um corpo contendo o nome do usuário e sua respectiva senha, caso contrário, um erro com o código 400 (Bad Request) será sinalizado com uma mensagem requerendo essas duas informações. Com esse corpo, uma verificação será feita realizando uma busca no Banco

de Dados com o intuito de verificar se o nome de usuário corresponde a algum usuário na tabela sys_users, e armazena os dados desse usuário em uma constante, caso contrário, outro erro de código 400 será sinalizado ao cliente da requisição.

```
const VerifyUserForUsername = async (dados) => {
  return await app
    .db("sys_users")
    .where({ username: dados.username })
    .first(); //Implementação de uma consulta ao banco de dados passando o
}; //corpo da requisição como parâmetro
```

Com os dados do usuário em questão armazenados numa constante, outras duas verificações serão feitas: se o atributo “flag” do usuário for 0 (inativo), e se a comparação da senha passada no corpo criptografada for igual a senha armazenada nessa consulta. Ambas validações retornando os mesmo tipos de erro citados anteriormente.

Passado por todas essas validações, o código deve, então, retornar ao cliente, um objeto em notação JSON contendo as seguintes informações:

- O ID do usuário;
- O email do usuário;
- O tipo do usuário (se é usuário proprietário ou padrão);
- Um token gerado pela codificação JWT;
- O horário em que foi gerado o Token, convertida em segundos (dividido por 1000);
- O horário em que o token expira, que corresponde ao seu valor somado a 259200 (duzentos e cinquenta e nove mil e duzentos).

```
    const isMatch =  bcrypt.compareSync(req.body.password, user.password);
    if(!isMatch) return res.status(400).send('Usuário ou Senha incorretos');

    const tokenNow = Math.floor(Date.now() / 1000);
    const playload = {
        id: user.id,
        email: user.email,
        username: user.username,
        type: user.type,
        issuered_at: tokenNow,
        expire_at: tokenNow + (60 * 60 * 24 * 3)


    }
    let tokenEncode = jwt.encode(playload, auth);
    res.json({ //Convertendo em notação JSON e enviando o objeto
        ...playload,
        token: tokenEncode
    });
}
```

### 1.2. Validação de token

A função que valida o token no BackEnd, o recebe no corpo da requisição de caminho "/v1/validateToken". Essa função lança uma invalidação para o cliente no caso de este corpo tiver sido enviado vazio, ou se a data de expiração do token, convertida para milissegundos, ter atingido seu limite de 259200s (duzentos e cinquenta e nove mil e duzentos segundos).

O retorno será em dado booleano, com valor “true” para token validado, e “false” para não validado. Este EndPoint deve ser chamado para verificações temporais de sessão do usuário logado na plataforma.

```
const validateToken = async (req, res) => {
    const userData = req.body || false; //objeto da requisição
    try {
        if(userData) {
            let token = jwt.decode(userData.token, auth); //decodifição do token por meio da senha
            if (new Date(token.expire_at * 1000) > new Date()) {
                return res.send(true);
            }
        }
    } catch (error) {
        //problema no token
    }
    res.send(false);
```

### 1.3. Autenticação por requisição

Para as próximas requisições do sistema, a função de autenticação será chamada previamente. Ela se encontra no arquivo passport.js, o qual faz referência a tecnologia que será utilizada para realização desse procedimento[1].

O Passport irá autenticar fazendo uso de uma estratégia JWT, que irá nutrir as informações do usuário armazenado, fazendo uma busca por seu ID. A estratégia utilizada recebe uma função de retorno que busca os usuários cadastrados no banco pelo ID passado pelo usuário em sessão, e verifica o resultado, retornando um erro de autenticação caso essa busca não aconteça (ignorando-se as razões que levam a isso), ou caso essa busca não encontre resultados.

## 2. Rotas Simples

Nesta sessão serão abordadas as rotas que tratam de objetos cujas tabelas possuem entidades que podem possuir relacionamento de até 1:N (um para N), mas nunca de N:N (muitos para muitos). O Banco de Dados segue o modelo relacional, e muitas das vezes, será comum presenciarmos colunas na tabela com a denominação "id" no final do objeto que queremos referenciar. Será esse o padrão de referência.

Frente a isso, também será verificado comandos SQL responsáveis por interligarem as tabelas por meio de seus ID’s, como os comandos LEFT JOIN e INNER JOIN, para que seja retornado outros atributos que serão utilizados como o nome do objeto do ID referenciado. Preferencialmente, os JOINS devem ser referenciados sempre pelo atributo ID.

## 2.1. Requisição verbo GET

Requisições do tipo GET sinalizam a intenção do cliente que mandou a requisição de obter dados do Banco de Dados. A função desse tipo chama métodos que enviam uma resposta para o usuário contendo os objetos modelados da maneira que for de melhor utilidade para a representação gráfica dos mesmos.

Em alguns casos, o EndPoint irá retornar uma resposta processada contendo dados de outras tabelas para o cliente, mapeados por meio de seus ID 's, como citado anteriormente. Muito utilizado principalmente pelas rotas de relatórios.

```
return await app.db
  .select("a.*", "b.flag") //mostrando todos os dados da tabela 'sys_dashboard_analytic', junto como a coluna 'flag' de sua fila associada
  .from("sys_dashboard_analytic as a")
  .leftJoin("ast_queues AS b", "a.queue_id", "b.id") //mapea a tabela 'sys_dashboard_analytic' com a tabela 'ast_queues'
  .leftJoin("sys_user_has_ast_queue as c", "a.queue_id", "c.queue_id")
  .where("c.user_id", params.userid)
  .where("b.flag", 1)
```

Segue uma tabela da requisição GET de cada EndPoint: Rotas simples - requisição via GET

## 2.2. Requisições verbo POST

Requisições do tipo POST sinalizam a intenção do cliente que mandou a requisição de inserir dados do Banco de Dados. A função desse tipo chama métodos que validam os dados que o cliente enviou para serem incluídos em um registro no banco, e enviam respostas de erro caso valores passados apresentem irregularidades.

No diretório ‘src/lib’, existe um arquivo de nome ‘validation.js’ que implementam funções validadoras. Elas devem ser chamadas em laços do tipo Try Catch, e devem receber uma mensagem de erro para serem lançadas ao cliente caso os dados passem por essa validação de erro.

```
const save = async (req, res) => {
    const agent = { ...req.body}
    if(req.params.id){
        agent.id = req.params.id
    }
    try {
        equalsOrError(agent.password, agent.confirmPassword,'Senha nao confere'); //Se o dado de senha for diferente de sua confirmação, será retornado a mensagem 'Senha nao confere' para o usuário
        existsOrError(agent.name, 'Nome nao informado');
        existsOrError(agent.agent , 'ramal nao informado');
        existsOrError(agent.password , 'Senha nao informado');
        existsOrError(agent.confirmPassword , 'Confirmação de senha invalida');

        const agentFromDB = await  agents.VerifyAgentByBranch(agent);


        if(!agent.type){
          agent.type = 'human';
        }

        if(typeof agent.flag == 'undefined'){
          agent.flag = 1
        }
        if(!agent.id){
          notExistsOrError(agentFromDB, "Este Ramal ja está cadastrado");
        }
    } catch(msg) {
        return res.status(400).send(msg)
    }
```

```
module.exports = (app) => {
  function existsOrError(value, msg) {
    if (!value) throw msg; //Lança a mensagem de erro quando o valor passado for nulo
    if (Array.isArray(value) && value.length === 0) throw msg; //Lança a mensagem de erro quando o valor passado não for uma matriz e não possuir elementos
    if (typeof value === "string" && !value.trim()) throw msg; //Lança a mensagem de erro quando o valor passado for um dado do tipo String sem caracteres
  }

  function equalsOrError(paramsA, paramsB, msg) {
    if (paramsA !== paramsB) throw msg; //Lança a mensagem de erro quando o primeiro parâmetro de validação for diferente do segundo
  }
```

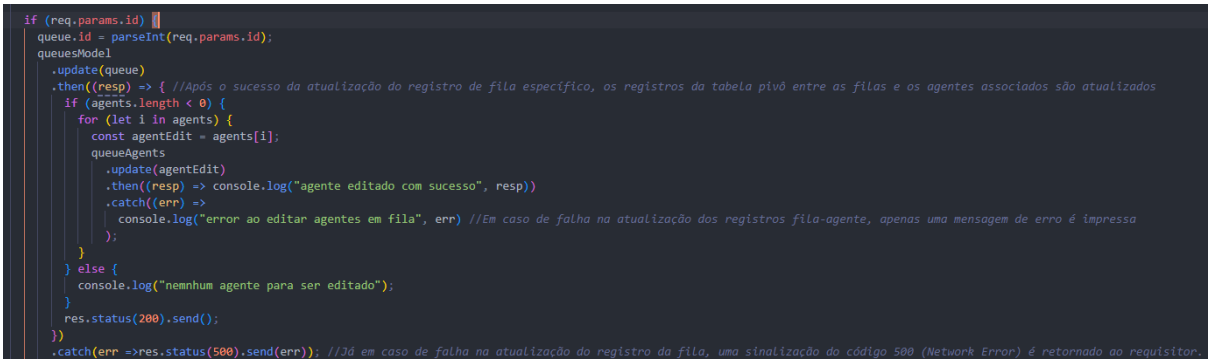
Segue uma tabela da requisição POST de cada EndPoint: Rotas simples - requisição via POST

### 2.3 Requisições verbo PUT

Requisições do tipo PUT sinalizam a intenção do cliente que mandou a requisição de editar registros já existentes no Banco de Dados. A função desse tipo geralmente é declarada nas mesmas funções da sessão controladora de buscas ao banco de dados, passando pelos mesmos processos de validação dos parâmetros enviados na requisição.

A única diferença é que esses métodos devem ser passados obrigatoriamente com um identificador (o ID do objeto), e deve retornar apenas o objeto cujo ID for igual ao passado por parâmetro na requisição. Na implementação da consulta ao banco, obrigatoriamente haverá o uso do comando WHERE, para comparar os ambos ID 's.

No caso abaixo, é apresentado um modelo especial de método de atualização de registro, onde um outro comando é chamado após o sucesso do objeto das filas, que é justamente o de atualização da tabela pivô com os agentes. É importante notar que existe uma ordem de atualização de tais registros, porém não há uma dependência, sendo necessário apenas que o método de edição de filas funcione para que o status de sucesso seja respondido ao cliente requisitor.



```
if (req.params.id) {
  queue.id = parseInt(req.params.id);
  queuesModel
    .update(queue)
    .then((resp) => { //Após o sucesso da atualização do registro de fila específico, os registros da tabela pivô entre as filas e os agentes associados são atualizados
      if (agents.length < 0) {
        for (let i in agents) {
          const agentEdit = agents[i];
          queueAgents
            .update(agentEdit)
            .then((resp) => console.log("agente editado com sucesso", resp))
            .catch((err) =>
              console.log("error ao editar agentes em fila", err) //Em caso de falha na atualização dos registros fila-agente, apenas uma mensagem de erro é impressa
            );
        }
      } else {
        console.log("nemnhum agente para ser editado");
      }
      res.status(200).send();
    })
    .catch(err =>res.status(500).send(err)); //Já em caso de falha na atualização do registro da fila, uma sinalização do código 500 (Network Error) é retornado ao requisitor.
```

Segue uma tabela da requisição PUT de cada EndPoint: Rotas simples - requisição via PUT

### 2.4. Requisições verbo DELETE

Requisições do tipo DELETE sinalizam a intenção do cliente que mandou a requisição de remover registros já existentes no Banco de Dados. Os métodos devem ser passados obrigatoriamente com um identificador (o ID do objeto) que se deseja realizar a exclusão no banco, e retornam apenas resultados de sucesso (200) ou falha (500), se tratando de funções bem simples no sistema, geralmente não ocupando mais que 6 linhas de código.

As rotas que possuem o método de deleção sempre serão procedidas por um parâmetro identificador em seu caminho (path), representado por dois pontos precedido por

um nome que será utilizado para nomear a constante que armazena esse dado identificador.

```
const remove = async (params) => {
    return await app.db("sys_servers").where({id: params.id}).del(); //o método de deleção de um registro não necessita de especificação por colunas da tabela, visto que só a coluna "ID" importa
}
```

## 3. Rotas de agrupamento de objetos

As rotas de agrupamentos de objetos tratam dos que possuem atributos tão complexos que foram necessárias novas tabelas de relacionamento com seus detalhes, transformando seus objetos em verdadeiros agrupamentos de outros pequenos objetos que podem, ou não, pertencer a outro grupo.

### 3.1. Agrupamento de itens correlacionados

Os agrupamentos que possuem itens dependentes, ou seja, que só podem existir se associados a um agrupamento apenas, possuem, em seu caminho URL, a denominação “/details”, e há também caminhos que carregam outros parâmetros atualmente inutilizados. Os objetos que possuem esses prolongamentos são Rotas, Tarifadores e Bloqueadores de DDD.

Para estas e outras rotas de relacionamento de objetos do banco, a função de editar (PUT) irá se tratar, na verdade, de um método de deleção seguido de um de inserção seguindo parâmetros específicos para cada relacionamento. O método PUT é usado apenas por sinalização, não se referindo a uma edição de dados em si, visto que é muito mais viável, pensando em inserções conjuntas, excluir todos os registros que correspondem ao objeto que irá ser atribuído e inserir uma lista de itens correspondente.

Um destaque importante se deve à validação dos detalhes dos Bloqueadores de DDD quando se acessa a rota no verbo PUT. O corpo da requisição deve ser inserido em um laço iterador para que, de cada Item, seja feito uma consulta na tabela "sys_DDDblockers_details", armazenando em uma lista, os bloqueadores que possuam o mesmo DDD, o mesmo dia da semana, a mesma data e estejam associados a agrupamentos diferentes, e então, para cada item desta lista, uma validação é feita para verificar se cada data inicial e data final da lista está dentro da faixa horária composta pela data inicial e data final do item do corpo da requisição iterado, e vice-versa. As faixas de horário são construídas por meio do plugin Moment e sua extensão “moment-range”, como mostrada abaixo na função validadora.

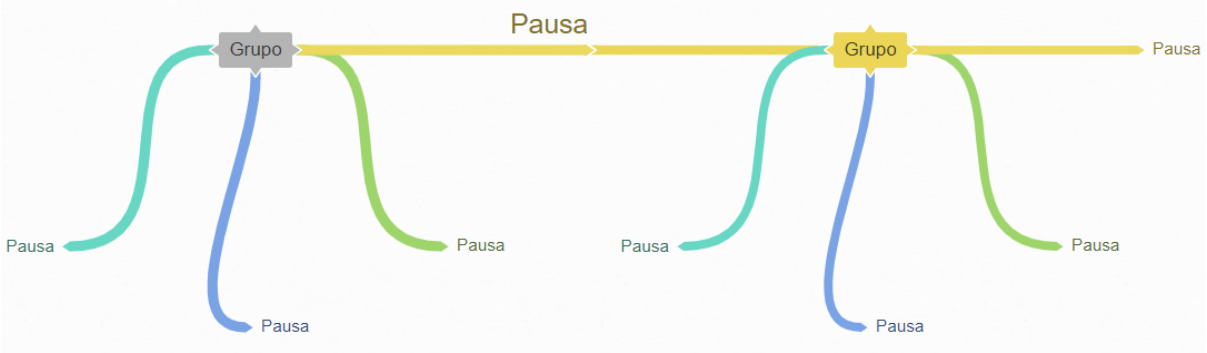
```
function notInRangeOrError(rangeOut, rangeIn, msg) { //Passado por parâmetro: dois objetos com os atributos de data inicial e data final e uma mensagem de erro a ser lançada para o cliente.
  const Moment = require("moment");
  const MomentRange = require("moment-range");

  const moment = MomentRange.extendMoment(Moment); //extendendo a função criadora de faixa horária

  //criação das faixas de horário para cada objeto criado. A função da biblioteca responsável chama-se "range".
  let periodIn = moment().range(rangeIn.init_hr, rangeIn.fnl_hr)
  let periodOut = moment().range(rangeOut.init_hr, rangeOut.fnl_hr)

  if (periodIn.contains(rangeOut.init_hr)) throw msg
  if (periodIn.contains(rangeOut.fnl_hr)) throw msg
  if (periodOut.contains(rangeIn.init_hr)) throw msg
  if (periodOut.contains(rangeIn.fnl_hr)) throw msg
}
```

Em caso de cada faixa de horário conter a data de início ou a data final de cada item do tipo objeto passado para esta função, a mensagem também passada por parâmetro será lançada para o cliente da requisição. Caso nenhum erro seja constatado, então o código segue para o procedimento de edição citado no início deste capítulo.

### 3.2. Agrupamento de itens não correlacionados

Os agrupamentos que possuem itens independentes, ou seja, que só podem existir sem estarem associados a um agrupamento, abrange apenas o objeto de Pausas.

As pausas possuem os atributos: nome, officer, produtiva, horário de alerta, horário limite, um ícone associado e status. Cada grupo de pausas possui um número de pausas atribuídas, com pausas que podem estar pertencendo a outro grupo.




Os métodos chamados pelas rotas fazem as inserções da mesma forma que as rotas que tratam de agrupamento de objetos correlacionados.

## 4. Rotas de relacionamento entre objetos

As rotas de relacionamento entre objetos tratam as relações do tipo N:M (muitos para muitos) entre os objetos do Banco de Dados do sistema WSuite. Essas rotas possuem as mesmas implementações para cada verbo da requisição que as rotas simples porém com algumas especificidades a mais. Os objetos que possuem relacionamentos de N:M são as Filas e os Usuários do sistema.

### 4.1. Relacionamento de filas

As filas são o objeto de maior atribuição de outros objetos do sistema, por isso merecem maior atenção quanto ao seu roteiro de inserção, validação, mapeamento de objetos e retorno de dados. Os procedimentos seguem os padrões citados anteriormente, porém seguem algumas especificidades.

Para se atribuir um mailing a uma fila, por exemplo, a tabela pivô “ast_queue_dialer” atribui o ID de um mailing com um ID de uma fila, e a ordem de inserção deve priorizar essa tabela em detrimento da tabela principal de mailing, que também possui uma coluna para representar o ID da fila associada, no entanto, apenas a impossibilidade de inserção na tabela pivô deve retornar um erro, como nos mostra o código abaixo.

```
if (req.params.id) {
  mailing.dialer_id = req.params.id;
  mailingQueue //implementação dos métodos na tabela pivô
    .update(mailing) //Atualiza os dados na tabela pivô
    .then((_) => { //com o sucesso na execução do método de atualização
      delete mailing.status;
      mailings.update(mailing).then((_) => { //atualiza o dado referente a fila na tabela de mailing
        res.status(200).send();
      });
    })
    .catch((err) => res.status(500).send(err));
} else {
  return res.status(400).send("requisição sem id");
}
```

Do mesmo modo, os caminhos URL seguem sempre uma sequência passando por parâmetro primeiro o objeto ao qual se quer atribuir, e, posteriormente, o objeto que se vai atribuir sempre por meio do seu ID. A tabela de mailing (dialer) se relaciona com a fila por meio de seu ID.

| | | | | | | |
|---|---|---|---|---|---|---|
| 1 | id | BIGINT | 20 | ☑ | ☐ | ☐ |
| 2 | queue_id | BIGINT | 20 | ☑ | ☐ | ☐ |
| 3 | user_id | BIGINT | 20 | ☑ | ☐ | ☐ |
| 4 | mailing_date | DATE | | ☐ | ☐ | ☐ |
| 5 | name | VARCHAR | 80 | ☐ | ☑ | ☐ |
| 6 | file | VARCHAR | 255 | ☐ | ☑ | ☐ |
| 7 | quantity | INT | 10 | ☑ | ☐ | ☐ |
| 8 | flag | TINYINT | 4 | ☐ | ☑ | ☐ |
| 9 | import | TINYINT | 4 | ☐ | ☐ | ☐ |
| 10 | copy | TINYINT | 4 | ☐ | ☐ | ☐ |
| 11 | count_rows | INT | 10 | ☑ | ☑ | ☐ |
| 12 | count_rows_virgin | INT | 10 | ☑ | ☑ | ☐ |

Para se atribuir uma ou mais Listas Negras a uma fila, o caminho URL “/v1/queue/blacklists/:id” (necessário o envio de ID, que será o identificador da fila ao qual se quer atribuir/obter dados de lista negra) chamará métodos que operam sobre a tabela “sys_blacklist_x_queues” responsável por fazer o relacionamento entre essas duas partes. Uma fila pode não possuir listas negras associadas, portanto, o script de edição é composto pela invocação de um método de exclusão na tabela de todos as listas negras associadas ao ID da fila em questão, procedido pelo método de inserção

```
const blacklists = req.body;
const queue_id = req.params.id;

for (let blacklist in blacklists) {
  delete blacklists[blacklist].filename;
}

await queueBlacklistsModel //Cada vez que receber um corpo vazio, apenas excluir todos os registros da tabela onde o ID da fila for igual ao passado
  .remove(queue_id) //por parâmetro
  .then((_) => {
    console.log(
      "Blacklists da fila de ID " + queue_id + " deletados com sucesso"
    );
    if (blacklists.length > 0) {
      queueBlacklistsModel.insert(blacklists);
    }
    res.status(204).send();
  })

  .catch((err) => res.status(500).send(err));
```

Esse modelo de edição também se repete para a associação de um roteiro a uma fila, no entanto, a tabela pivô “sys_qss” faz pivô também com servidores, e estes, por sua vez, devem possuir o atributo “tipo”(type) como “robô” (robot), portanto, antes de incrementar no banco de dados, será feito uma busca por todos os servidores que seguem essa regra, e deve-se modelar um objeto com base nessa busca para ser inserido na tabela pivô.

```
const queue_id = req.params.id;

await queueScript
  .remove(queue_id)
  .then((_) => {
    console.log(
      "Script da fila de ID " + queue_id + " deletados com sucesso"
    );
    if (req.body.script_id) {
      servers.selectWhereType("robot").then((res) => { //Busca por todos os servidores que sejam do tipo "robô"
        let params = [];
        for (let i in res) {
          params.push({ //modelando o objeto pivô a ser inserido
            queue_id,
            server_id: res[i].id,
            script_id: req.body.script_id,
          });
        }
        queueScript.insert(params);
      });
    }
    res.status(204).send();
  })
  .catch((err) => res.status(500).send(err));
```

### 4.2. Relacionamento de usuários

O objeto Usuários se refere aos atores do sistema, ou seja, todos aqueles que executam métodos ao banco de dados fazendo uso da interface digital do Wsuite. Cada usuário possui, como chaves primárias, seu identificador, um email, e um nome de usuário. Seu perfil indica suas ações autorizadas, sendo o perfil administrador o que dá plenos poderes aos usuários de acessar as rotas da API.

Para se atribuir uma fila a um usuário do sistema, o processo é feito sobre a tabela “sys_user_has_ast_queue”, seguindo o mesmo procedimento de exclusão e inserção para o verbo PUT citado anteriormente. Não há rota responsável para esse segmento, o método é feito no controlador de usuário, nos métodos de adicionar e editar usuário.

```
const userQueuesDelete = async (params) => {
  return await app
    .db("sys_user_has_ast_queue")
    .where({ user_id: params.id })
    .del(); //Comando de deleção
};
const usersQueuesInserir = async (userQueue) => {
  return await app.db("sys_user_has_ast_queue").insert(userQueue); //Método de inserção sobre a tabela pivô deve ser chamada dentro um laço para se inserir um registro de cada vez.
};
```

Também na função controladora, ao se criar um usuário atribuindo-lhe um ID de perfil, o usuário estará apto a visualizar as páginas do sistemas denominadas pelo objeto Páginas, da tabela referente “sys_pages”. Seus atributos informam quais das funções CRUD estão disponíveis para tal página.

```
module.exports = (app) => {
  const selectAll = async () => {
    return await app.db
      .select(
        "a.id as page_id",
        "a.name",
        "b.name as modulo_name",
        "a.create as add",
        "a.read",
        "a.update as edit",
        "a.delete"
      )
      .from("sys_pages as a")
      .innerJoin("sys_modules as b", "b.id", "a.modules_id");
  }; //função model de obtenção das páginas do Wsuite. Mostra também os módulos atribuídos a cada página.
  return {
    selectAll,
  };
};
```

No entanto, é somente o Perfil que informa quais métodos o usuário está autorizado a requisitar. A tabela “sys_perfils_pages”, então, mostra quais páginas o usuário associado aquele ID de perfil pode acessar, quais rotas ele pode requerer, quais rotas ele está autorizado a consumir, e quais os módulos.

O roteiro para salvar um perfil segue o seguinte algoritmo:

1) Verificar se o nome foi passado no corpo da requisição.
2) Verificar se a lista de páginas passadas está vazia, alegando que o usuário não se atentou a essa parte, visto que é ilógico um perfil sem poderes.
3) Verificar se um parâmetro identificador do perfil foi passado.
    - Caso tenha sido passado, os dados da tabela que trata do objeto Perfil (sys_perfils), em si, serão atualizados.
    - Caso contrário, será adicionado um novo registro com os dados de nome, descrição e o parâmetro de ativação (status) em booleano numérico do novo perfil.
4) Construir uma nova lista e nesta incluir cada dado objeto da lista de páginas passada no corpo da requisição, inserido junto como propriedade, o ID desse perfil.
5) Inserir essa lista na tabela “sys_perfils_pages”.

## 5. Rotas Especiais

Nesta sessão, serão abordadas as rotas de upload de arquivo mailing, e de layout de filas, rotas com procedimentos que não seguem o padrão escolhido para categorização dos EndPoints.

A rota de caminho "/v1/mailing/upload-arquivo", possui o método POST cuja função faz uso de um middleware nomeado como Multer, que é uma implementação de um plugin que envia arquivos para outros servidores.

```
multerUploader(req, res, (err) => { //envio do arquivo, capturando a requisição, emoldurando a resposta e um possível erro
  const mailing = { ...req.body };
  if (err) { //Tratando o retorno de erros no caso de algum capturado durante o processamento e envio do arquivo
    res.status(415).send({
      msg: err,
    });
  } else {
    if (req.file == undefined) {
      res.status(400).send({
        msg: "Error: No File Selected!",
      });
    } else {
      delete mailing.filename;

      mailing.file = req.file.filename;
      mailing.quantity = 0;
      mailings
        .insert(mailing)
        .then((resp) => {
          res.json({
            msg: "File Uploaded!",
            file: `uploads/${req.file.filename}`,
          });
        })
        .catch((err) => res.status(500).send(err));
    }
  }
});
```

Com o sucesso do envio do arquivo de mailing, passa-se ao roteiro normal de inserção no banco de dados sobre a tabela “dialer”.

Para a rota "/v1/layout/", seu único verbo de requisição indica para uma um arquivo na pasta “wsuite-api\src\models\v1” que será reservado para consultas ao banco de dados por meio de métodos do próprio SRGB, representados em “INFORMATION_SCHEMA”. Neste arquivo, a função que será chamada para esta rota chama-se “fields”, e ela busca as todas as cerca de 260 colunas da tabela “dialer_data” passada por parâmetro.

```
module.exports = (app) => {
    const fields = async (params) => {
      return await app.db.select('COLUMN_NAME').from('INFORMATION_SCHEMA.COLUMNS').where('TABLE_NAME', params.table_name)
    };
  return {
      fields,
  }
}
```

Futuramente, esse arquivo deve ser incrementado com novas funcionalidades que requerem informações sobre a estrutura do BD.

## 6. Referências


Front end: O que é, como funciona e qual a importância. **TOTVS**, 14 de jul. de 2021. Disponível em: https://www.totvs.com/blog/developers/front-end/#:~:text=Front%2Dend%3A%20o%20que%20%C3%A9,%2C%20sites%2C%20aplicativos%2C%20etc. Acesso em: 01 de jun. de 2022

Uma breve introdução sobre BCrypt. **Medium**, 10 de dez. de 2019. Disponível em: https://medium.com/reprogramabr/uma-breve-introdu%C3%A7%C3%A3o-sobre-bcrypt-f2fad91a7420. Acesso em: 01 de jun. de 2022

Como o JWT funciona. **Devmedia**. Disponível em: https://www.devmedia.com.br/como-o-jwt-funciona/40265. Acesso em: 01 de jun. de 2022

Autenticação em Node.js com Passport. **luiztools**, 11 de nov. de 2020. Disponível em: https://www.luiztools.com.br/post/autenticacao-em-node-js-com-passport/. Acesso em: 01 de jun. de 2022

Relacionamento 1-1, 1-N e N-N com Django. **Treinaweb**. Disponível em: https://www.treinaweb.com.br/blog/relacionamento-1-1-1-n-e-n-n-com-django#:~:text=No%20modelo%20relacional%2C%20existem%20tr%C3%AAs,N%2DN%20(muitos%20para%20muitos). Acesso em: 31 de mai. de 2022

Passport- JWT Documentation. Disponível em: http://www.passportjs.org/packages/passport-jwt/. Acesso em 01 de jun. de 2022.