Relatório do trabalho prático

Versão 1.0

Realizado por:

Samuel Cunha, 8160526
Jorge Moreira, 8160297
Marcelo Carvalho, 8160287

12/05/2020

Índice

Índice de Figuras

Datas importantes

Nomes	Data	Mudanças feitas	Versão
Jorge Moreira Marcelo Carvalho Samuel Cunha	15/05/2020	Foi feita uma Introdução do trabalho e a explicação do que foi feito até agora tais como: rotas, módulos, controllers, middlewares e decisões.	1.0

1. Introdução

Este trabalho consiste na criação de uma aplicação web que vá ao encontro de uma solução considerando os tempos atuais devido ao Covid-19. Esta aplicação tem como objetivo permitir pedidos de diagnóstico ao centro de análises que deverá processar o pedido por um técnico do centro de análises.

No pedido deve anotar informações relevantes tais como códigos de identificação, uma breve descrição sobre o teste/paciente, se o utilizador pertence a um grupo de risco, o estado do teste, o resultado do teste, as marcações dos testes e entre outros.

1.1 Milestone #1

Para o desenvolvimento do primeiro *milestone,* foram considerados as especificações e a elaboração de todos os serviços *REST* necessários para a aplicação com os *endpoints* desenvolvidos em *nodeJS*. Estes serviços foram testados na aplicação *Postman* tendo todos dado uma resposta positiva.

Foi também realizado a documentação da *API REST* em *Swagger*. O *Swagger* é uma poderosa ferramenta que ajuda a projetar, desenvolver, documentar e consumir serviços web *RESTful*. Foi feita a documentação das rotas e dos *models*.

Por fim também foi realizado um mecanismo de autenticação através de *JWT (JSON Web Token)*.

2. Pressupostos

Para este primeiro *milestone*, tendo em conta os requisitos gerais, foram realizados todos os *endpoints* necessários para a utilização da *api rest*, exceto duas funcionalidades:

- Agendamento automático;

- *Dashboards* e uso de gráficos, onde parecerão em conjunto com o número de testes realizados por dia, número de testes realizados por pessoa e números de pessoas infetadas;

O agendamento neste primeiro *milestone* foi feito de forma manual, e os *dashboards* serão apresentados no segundo *milestone*, com a parte de Angular do trabalho. Porém neste *milestone* será entregue a *api Rest* junto deste relatório, e alguns *mockups.* Foi também realizada a documentação *swagger* da *api* e o mecanismo de autenticação.

3. Interpretação do Enunciado, Models e Rotas escolhidas

Pretende-se que com este trabalho seja criada uma aplicação web que faça a gestão da realização de testes de *covid* para um centro de análises.

Como estava descrito no enunciado, percebemos que a *API* necessitava de uma gestão de utilizadores e de testes de *covid*, e por isso antes de passarmos à criação de *endpoints* e controladores, criamos um único *schema* para utilizadores e testes de *covid*, em que os utilizadores eram diferenciados pela role:

- ADM para administradores;

- TECH para técnicos;

- EXT para utilizadores externos.

Para os testes de *covid* foi criado o *schema* seguinte:

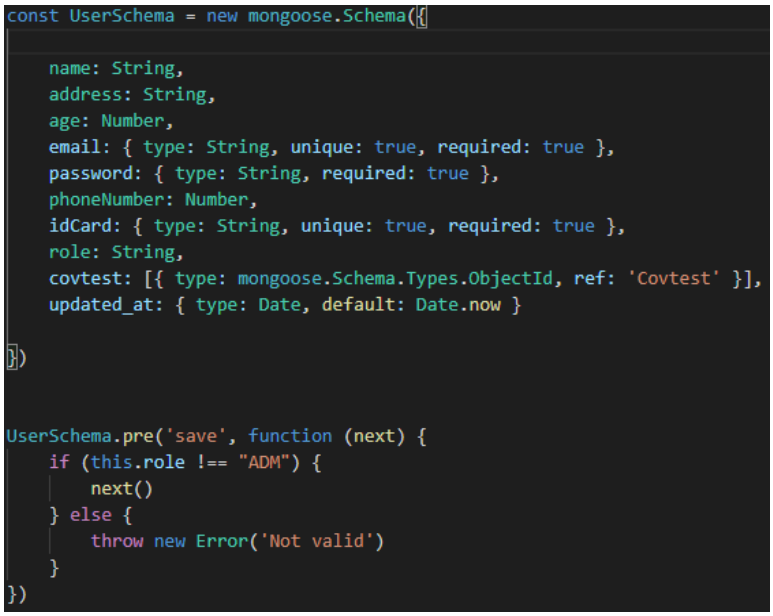
```
const CovtestSchema = new mongoose.Schema({

    code: { type: String, unique: true, required: true },
    description: String,
    userHistory: String,
    userStatus: String,
    riskGroup: String,
    riskJob: String,
    testStatus: String,
    testResult: String,
    resultFile: String,
    isTestDone: Boolean,
    schedule: Date,
    saude24: Boolean,
    user: { type: mongoose.Schema.Types.ObjectId, ref: 'User' },
    updated_at: { type: Date, default: Date.now }

})
```

Figura 1 - Ficheiro Covtest.js

Para os utilizadores foi criado o *schema* seguinte:



```
const UserSchema = new mongoose.Schema({

    name: String,
    address: String,
    age: Number,
    email: { type: String, unique: true, required: true },
    password: { type: String, required: true },
    phoneNumber: Number,
    idCard: { type: String, unique: true, required: true },
    role: String,
    covtest: [{ type: mongoose.Schema.Types.ObjectId, ref: 'Covtest' }],
    updated_at: { type: Date, default: Date.now }

})


UserSchema.pre('save', function (next) {
    if (this.role !== "ADM") {
        next()
    } else {
        throw new Error('Not valid')
    }
})
```

Figura 2 - Ficheiro User.js

Nota: admitimos que o Administrador vem por defeito na *API*, por isso criamos o utilizador antes de criar a função “UserSchema.pre” para impedir que algum *user* crie administradores.

Após a criação dos esquemas da base de dados, e analisando os requisitos funcionais, criámos *controllers* e rotas (ficheiro User-route.js) para os técnicos fazerem a gestão de utilizadores externos, à exceção de uma rota para listar todos os utilizadores (exclusiva ao *Admin*). Rota users/:

- users/userListExt -> Rota *get* que retorna todos os utilizadores externos. (Acessível também pelo administrador);
- users/userList -> Rota *get* que retorna todos os utilizadores do sistema. (Exclusivo do Admin);
- users/:id -> Rota *get* que retorna um utilizador externo com um *idCard* como parâmetro em id, *especifíco*;
- users/create -> Rota *post* para a criação de um utilizador externo;
- users/:id -> Rota *delete* para apagar um utilizador externo;
- users/:id -> Rota *put* para atualizar campos do utilizador externo;
- users/tests/:id -> Rota *put* para associar testes de *covid* a um utilizador;
- users/tests/:id -> Rota delete para remover testes de *covid* da lista de testes de um utilizador;

```
//List all users
router.get('/userList', authorize(['ADM']), function (req, res) {
    User.listAllUsers(req, res)
})

//get specific user by ID
router.get('/:id', authorize(['TECH']), function (req, res) {
    User.findOneUser(req, res)
})

//Create user
router.post('/create', authorize(['TECH']), function (req, res) {
    User.createUser(req, res)
})

//delete user
router.delete('/:id', authorize(['TECH']), function (req, res) {
    User.deleteUser(req, res)
})

//update user
router.put('/:id', authorize(['TECH']), function (req, res) {
    User.updateUser(req, res)
})
```

Figura 3 - Ficheiro User-route.js (excerto)

Foram criadas rotas para o administrador fazer a gestão de técnicos (tech-route.js):

- techs/Create -> Rota *post* para criar técnicos;
- techs/techList -> Rota *get* que retorna a lista de técnicos;
- techs/:id -> Rota *delete* que apaga um técnico com um *idCard* específico;
- techs/:id -> Rota *put* que atualiza um técnico com um *idCard* específico;
- techs/:id -> Rota *get* que obtém um técnico com um *idCard* específico;

```
//create tech user
router.post('/create', authorize(['ADM']), function (req, res) {
    Tech.createUserTech(req, res)
})

//list all users in role tech
router.get('/techList', authorize(['ADM']), function (req, res) {
    Tech.listUserTech(req, res)
})

//delete a specific tech user
router.delete('/:id', authorize(['ADM']), function (req, res) {
    Tech.deleteUserTech(req, res)
})

//update specific tech user
router.put('/:id', authorize(['ADM']), function (req, res) {
    Tech.updateUserTech(req, res)
})

//get specific tech user
router.get('/:id', authorize(['ADM']), function (req, res) {
    Tech.findOneUserTech(req, res)
})

module.exports = router;
```

Figura 4 - Ficheiro tech-route.js

Para além destas rotas, o *admin* consegue mudar a sua *password* e obter a sua informação (Adm-route.js):

- admin/ -> Rota *get* que obtém a informação do *admin*;
- admin/changePass/:id -> rota *put* que altera a passe do *admin* passando *o idcard* como *id;*

```
var express = require('express');
var router = express.Router();
var Admin = require("../controllers/AdmController");
const authorize = require('../middleware/authorize')

//get Admin
router.get('/', function (req, res) {
    Admin.getAdmin(req, res)
})

//create admin -> ROUTE CREATED ON PURPOSE FOR ADMIN TESTS!
/*
router.post('/', function (req, res) {
    Admin.createUser(req, res)
})
*/

//Update Admin password
router.put('/changePass/:id', authorize(['ADM']), function (req, res) {
    Admin.updatePassword(req, res)
})


module.exports = router;
```

Figura 5 - Ficheiro Adm-route.js

Existem 3 rotas para tratar da sessão dos utilizadores (Session-route.js):

- /login -> Rota *post* de login;
- /me -> Rota *get* que obtém o utilizador que fez login (com esta rota, cada utilizador poderá depois ver a sua informação na *dashboard*, por exemplo os utilizadores externos poderão depois ver os próprios testes de *covid*);
- /logout -> rota *post* de *logout*;

```
const SESSION_EXP = 600000
const {
    JWT_SECRET = 'this is for development'
} = process.env

sessionRouter.post('/login', async (req, res, next) => {

    const user = await User.findOne({ email: req.body.email });
    bcrypt.compare(req.body.password, user.password, (err, result) => {
        if (err) {
            return res.status(401).json({
                message: "Auth failed"
            });
        }
        if (result) {
            const jwtToken = jwt.sign(JSON.stringify(user), JWT_SECRET)
            res.cookie(
                'session',
                jwtToken,
                {
                    expires: new Date(Date.now() + SESSION_EXP),
                    httpOnly: true
                }
            )
        }
        res.json(user)

    })
})
```

Figura 6 - Ficheiro Session-route.js (excerto)

Para a gestão de pedidos, criamos rotas para listagem, em que algumas delas contém filtros específicos para cada utilizador (Covtest-route.js):

- covtests/testList -> Rota *get* exclusiva ao administrador e ao técnico para obterem a listagem de todos os testes;
- covtests/pending -> Rota para os técnicos obterem testes com status *"pending";*
- covtests/positive -> Rota para os técnicos obterem testes com resultados positivos;
- covtests/negative -> Rota para os técnicos obterem testes com resultados negativos;
- O Administrador pode obter o número de testes realizados com a rota *"covtests /count"*, o que achamos também necessário para mais tarde colocar na dashboard;
- covtests/listTest/:id -> O técnico pode ver testes de *covid* associado a um utilizador.
- covtests/schedule/:id -> O técnico pode alterar a data do teste de *covid* manualmente

Os técnicos podem fazer upload de PDFs para anexar a um teste de *covid* assim como podem efetuar o download. Quanto aos utilizadores externos, só podem fazer o download:

- Covtests/upload/:id;
- Covtests/download/:id;
- Nota: em ambas as rotas, o id é o *objectId* do teste que entre como parâmetro;

O utilizador externo pode criar o teste de *covid* (efetuando o pedido), e este irá ser associado automaticamente ao utilizador. O teste de *covid* também tem um código que é gerado automaticamente com *"uniqid":*

- Covtests/create/:id -> Rota *post* que cria o teste associando o utilizador correspondente ao *objectId* que entre como parâmetro;

Por fim o técnico pode editar o estado do teste, o estado do utilizador no teste e o resultado do teste respetivamente:

- update/testStatus/:id
- update/testUserStatus/:id
- update/testResult/:id
- Nota: o código gerado aleatoriamente com *uniqid* entre como parâmetro no id

```
//List tests
router.get('/testList', authorize(['ADM']), function (req, res) {
    Covtest.listTests(req, res)
})

//Create covid test with a id from  user
router.post('/create/:id', authorize(['EXT']), function (req, res) {
    Covtest.createTest(req, res)
})

//update test status
router.put('/update/testStatus/:id', authorize(['TECH']), function (req, res) {
    Covtest.updateTestStatus(req, res)
})

//update user test status
router.put('/update/testUserStatus/:id', authorize(['TECH']), function (req, res) {
    Covtest.updateTestUserStatus(req, res)
})

//update test result
router.put('/update/testResult/:id', authorize(['TECH']), function (req, res) {
    Covtest.updateTestResult(req, res)
})
```

Figura 7 - Ficheiro Covtest-route.js (excerto)

Por fim concluímos todas as definições das rotas para o uso da *apiRest*. As rotas estão também documentadas e funcionais na documentação *swagger* da *api*, em que está descrito o que será enviado no *body* e nos parâmetros de cada rota.

Os *controllers* irão ser explicados mais à frente.

4. Administrador

O administrador é um tipo de utilizador que é criado antes da utilização da aplicação. Para criar o administrador é necessário remover de comentário o seguinte código presente no ficheiro *Adm-route.js,* pois esta é a rota necessária á criação do admin:

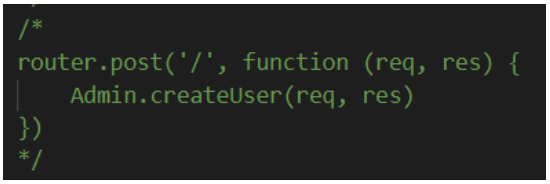


Figura 8 - admin (1)

De seguida, no *AdmController.js,* remove-se de comentário a função de criação de admin:

Figura 9 - admin (2)

No final, é necessário ir ao schema de utilizadores e comentar o código seguinte, que impede a atribuição da role “ADM”:

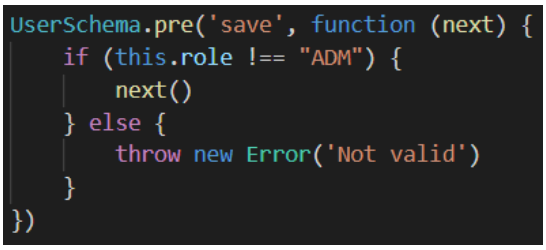


Figura 10 - admin (3)

Posto isto, é possível criar admins a partir da rota /admin/ a partir do postman.

5. Características específicas do projeto

Neste tópico estão todas as características específicas do projeto.

5.1. Ficheiro package.json

Este é um ficheiro importante para o projeto. É nele onde são declaradas todas as dependências do mesmo. Na seguinte imagem podemos ver as dependências declaradas:

Figura 11 - Dependências package.json

As dependências que podemos ver são:

- bcrypt: para encriptar passwords (irá ser usado na parte de autenticação);

- cookie-parser: analisa cookies e coloca informações sobre o objeto “req” no middleware. Incluí funcionalidades para fazer o parse de cookies nos pedidos http para o servidor;

- cors: o cors é um pacote node.js para fornecer um middleware do Connect / Express que pode ser usado para ativar o CORS com várias opções. Permite o compartilhamento de recursos de origem cruzada;

- dotenv: é um módulo com dependência zero que carrega variáveis de ambiente de um ficheiro “.env” para um “process.env”

- ejs: é um template engine disponível para NodeJS. Difere pela sintaxe usada para criar o conteúdo dinâmico nas páginas. Permite o uso direto de código javascript usando tags especificas definidas na sua especificação;

- express: Permite a criação de um servidor em NodeJS. Tem uma arquitetura simples e agnóstica do ponto de vista de padrões de software. Permite filtrar pedidos ao servidor por método (GET, POST, etc…) e por rota;

-jsonwebtoken: JSON Web Token (JWT) é um open standard (RFC 7519) que define um método compacto e autocontido para transmitir com segurança informações entre as partes num objeto JSON;

- mongoose: módulo para comunicar com base de dados MongoDB;

- mongoose-immutable-plugin: permitir que os esquemas tenham propriedades imutáveis;

- node-fetch: A API Fetch fornece uma interface JavaScript para acessar e manipular partes do pipeline HTTP, tais como os pedidos e respostas. Ela também fornece o método global fetch() que fornece uma maneira fácil e lógica para buscar recursos de forma assíncrona através da rede;

- swagger-ui-express: módulo que ajuda com a tarefa de documentar a API;

- uniqid: cria IDs exclusivos com base no horário atual, no processo e no nome da máquina;

- nodemon: é uma ferramenta que ajuda a desenvolver aplicações baseadas em node.js. Reinicia automaticamente a aplicação quando são detetadas alterações nos ficheiros.

5.2. Controllers

O Controlador (*controller*) envia comandos para o modelo para atualizar o seu estado. Antes de definir as rotas, primeiro criaremos todas as funções de retorno que elas chamarão. Os retornos serão armazenados em módulos "controladores" separados para *User* e *Covtest*. Os controladores são uma parte muito importante no padrão MVC.

5.2.1. AdmController

No ficheiro “AdmController.js” podemos ver os *controllers* para os administradores.

Na imagem seguinte podemos ver os controllers feitos:

```
//Update Admin password (admin)
AdmController.updatePassword = async (req, res) => {
    const encryptedPass = bcrypt.hashSync(req.body.password, 10);

    const newData =
    {
        password: encryptedPass
    }

    await User.findOneAndUpdate({ idCard: req.params.id, role: "ADM" }, newData);
    const result = await User.find({ idCard: req.params.id, role: "ADM" })
    res.json(result)
}

//find the admin (admin)
AdmController.getAdmin = async (req, res) => {
    const result = await User.findOne({role: "ADM"})
    res.json(result)
}
```

Figura 12 - Ficheiro AdmController.js (excerto)

updatePassword: tendo em conta o que foi pedido no enunciado, quanto à gestão de utilizadores, deve ser possível editar a password do administrador. Para isso iremos encriptar a password e encontrar um utilizador cuja sua role seja “ADM”, ou seja, um administrador e fazer o update da palavra passe. Depois é passado o resultado.

getAdmin: encontrar um utilizador cuja role seja “ADM”, ou seja, um administrador para a exibição do seu perfil.

5.2.2. CovtestController

No ficheiro “CovtestController.js” podemos ver os controllers para os testes.

Pelo que foi absorvido pelo enunciado, segue uma explicação sobre cada um. Na imagem podemos ver um excerto:

```
//list tests (admin)
CovtestController.listTests = async (req, res) => {
    const testList = await Covtest.find().
        populate('user', ['name', 'idCard'])
    res.json(testList);
}

//create test with associated user (user)
CovtestController.createTest = async (req, res) => {
    const targetUser = req.params.id;

    const randomCode = uniqid.process('', '-Covtest')
    const newData =
    {
        ...req.body,
        code: randomCode,
        //associate the user that created the test
        user: targetUser,
        testStatus: "pending"

    }

    //create test
    const test = await Covtest.create(newData)
```

Figura 13 - Ficheiro CovtestController.js (excerto)

- listTests: o populate é usado para juntar documentos. Vai buscar à coleção os users com nome e “idCard” para popular os utilizadores nos testes e irá retornar esse teste.

- createTest: como o próprio nome diz, irá criar um teste. Para isso, cada teste irá ter o seu id único seguido de “-Covtest”. Irá associar o teste ao utilizador que o criou e coloca o estado como pendente. De seguida o teste é criado e posto na lista do utilizador (“EXT” porque é um utilizador) e de seguida é “populado” para mostrar o teste;

- updateTestUserStatus: quem pode fazer as atualizações dos testes são os técnicos. Neste caso, para fazer a atualização do estado do utilizador, é passado um novo dado para “userStatus”. Para isso é usado o método “findOneAndUpdate” para encontrar o teste e fazer o seu update e retornar o resultado;

- updateTestStatus: quem pode fazer as atualizações dos testes são os técnicos. Tal como no anterior, para fazer a atualização do estado do teste, é passado um novo dado para “testStatus”. É usado o método “findOneAndUpdate” para encontrar o teste e fazer o seu update, retornando de seguida o resultado;

- updateTestResult: para atualizar o resultado do teste é fazer praticamente o mesmo que foi feito dos dois updates anteriores;

- listUserTests: como o próprio nome indica, irá apresentar uma lista de testes de um utilizador. Para isso, teremos que encontrar o utilizador pelo seu id;

- listPend: buscar uma lista de testes em que o estado do teste está como pendente (pending);

- listPos: buscar uma lista de testes em que o resultado do teste está dado como positivo (Positive);

- listNeg: buscar uma lista de testes em que o resultado do teste está dado como negativo (Negative);

- countTest: irá contar o número de testes através do método “estimatedDocumentCount()”.

5.2.3. FileController

No ficheiro “FileController.js” podemos ver os controllers para o ficheiro dos testes.

Pelo que foi pedido no enunciado, quando obtidos os resultados deve ser possível registar o resultado clínico na ficha do pedido do utilizador anexando um ficheiro (pdf) com os resultados clínicos e adicionando o resultado final ao pedido.

Na imagem seguinte podemos ver um excerto, seguido depois da explicação do que foi implementado:

```
//Upload File (tech)
FileController.upload = async (req, res) => {
    if (!req.files || Object.keys(req.files).length === 0) {
        return res.status(404).send(`<h1>NO FILES UPLOADED</h1>`);
    }
    let sampleFile = req.files.file
    let uploadPath = "./public/uploads/" + sampleFile.name;

    //console.log(uploadPath)
        try {
            await Covtest.findOneAndUpdate({ code: req.params.id }, { resultFile: uploadPath })
            const result = await Covtest.find({ code: req.params.id })

            const rr = await Covtest.find({ code: req.params.id },'description' )
            console.log(rr)
            sampleFile.mv(uploadPath, function (err) {
                if (err)
                    return res.status(500).send(err);

                res.json(result);
            });
        }catch(err){
        console.log(err)
    }
}
```

Figura 14 - Ficheiro FileController.js (excerto)

upload: depois de termos o ficheiro teremos que o anexar, ou seja, fazer um upload e isso engloba alguns passos para o fazer. Primeiro temos que ver se estamos realmente a enviar um ficheiro e caso negativo, temos uma mensagem que nenhum ficheiro foi enviado (No Files Uploaded). De seguida iremos encontrar o teste a ser enviado e fazer o seu update com o caminho (path) do ficheiro. Ao enviar fazemos uma simples condição para o caso de haver algum erro. Caso contrário o ficheiro é enviado com sucesso;

download: o download do ficheiro é mais simples de implementar. Basta encontrar o ficheiro do teste a ser descarregado. Se esse ficheiro não for encontrado é enviada uma mensagem a com o erro 404 de não encontrado. Caso seja encontrado é feito o download do ficheiro com uma mensagem de “File downloaded”.

5.2.4. TechController

No ficheiro “TechController.js” podemos ver os controllers para a manutenção / gestão de técnicos, tal como pedido no enunciado. Quando à gestão dos técnicos, somente os administradores têm autorização, como já podemos ver anteriormente.

Na imagem vemos um excerto deste ficheiro e a explicação:

```
//create tech user (admin)
TechController.createUserTech = async (req, res) => {
    if (req.body.role == null) {

        //forçar os tecnicos nao terem testes de covid na bd
        if (req.body.covtest !== []) {
            req.body.covtest == []
        }

        const encryptedPass = bcrypt.hashSync(req.body.password, 10);
        const newData =
        {
            ...req.body,
            password: encryptedPass,
            role: "TECH"
        }
        const result = await User.create(newData);
        res.json(result);
    } else {
        console.log("User is technical by default");
        res.send()
    }
}

//delete tech user (admin)
TechController.deleteUserTech = async (req, res) => {
    const user = await User.findOne({ idCard: req.params.id })
    if (user.role === "TECH") {
        await user.remove()
        res.json(user)
    } else {
        console.log("User is not a tech")
        res.send()
    }
}
```

Figura 15 - Ficheiro TechController.js (excerto)

- createUserTech: como se sabe, não faz sentido os técnicos terem associados testes de covid na base de dados, visto que o problema está projetado para os utilizadores. Por isso começamos logo com uma verificação para forçar os técnicos a não terem testes de covid. De seguida é criada uma password encriptada e com a sua função (role) para “TECH” e criar o técnico ao enviar o resultado. Caso o utilizador a ser criado já seja um técnico é enviada uma mensagem a dizer que o utilizador já é um técnico por defeito;

- deleteUserTech: para proceder à eliminação de um técnico, é necessário encontrá-lo pelo seu “idCard”. Caso a sua função (role) seja técnico (TECH), é feito o remove do user e retornado o resultado. Caso não seja um técnico, é apresentada uma mensagem a dizer que o utilizador não é um técnico;

- updateUserTech: para atualizar um técnico, temos de ter a certeza que não alteramos a sua role, ou seja vamos forçar para que seja sempre um técnico. Tal como a criação do técnico, forçamos novamente para que não ter testes de covid na base de dados. Se for enviada uma palavra passe nova ela tem de ser encriptada. É procurado o utilizador com a role “TECH” pelo seu id e de seguida é enviado o resultado;

- listUserTech: muito simples de implementar, onde basta encontrar o utilizador cuja função seja “TECH” e enviar o resultado para a apresentação;

- findOneUserTech: também muito simples, onde o objetivo é buscar um técnico específico. Pode ser feito através de uma busca usando o método “findOne” para encontrar através do seu id e cuja role seja “TECH”. Caso não seja encontrado o técnico é enviada uma mensagem que não encontrado o técnico. Caso seja encontrado é enviado o resultado.

5.2.5. UserController

Por fim, no ficheiro “UserController.js” foram feitos os controllers para manipular os utilizadores, em que somente o administrador e técnicos têm autorização para executar, tal como já tínhamos visto.

Segue na imagem seguinte um excerto, seguido da explicação:

```
//list all users (admin)
UserController.listAllUsers = async (req, res) => {
    const list = await User.find().
        populate('covtest', 'code')
    res.json(list);
}

//list all External users (tech)
UserController.listExtUsers = async (req, res) => {
    const list = await User.find({ role: "EXT" }).
        populate('covtest', 'code')
    res.json(list);
}

//find one user (tech)
UserController.findOneUser = async (req, res) => {
    const result = await User.findOne({ idCard: req.params.id, role: "EXT" }).
        populate('covtest', 'code')
    res.json(result);
}
```

Figura 16 - Ficheiro UserController.js (excerto)

- listAllUsers: o populate vai juntar as coleção dos users e dos testes para apresentar um utilizador com um teste associado;

- listExtUsers: o mesmo que o anterior mas somente encontramos utilizadores (EXT);

- findOneUser: o mesmo que os anteriores mas temos que encontrar um utilizador especifico através do seu idCard cuja role seja “EXT”;

- createUser: se o utilizador não tiver nenhuma função associada, iremos atribuir uma password encriptada. De seguida é enviado o resultado com os novos dados;

- deleteUser: encontrar um utilizador pelo seu idCard cuja role seja “EXT” e proceder à sua eliminação através do método findOneAndDelete();

- updateUser: muito parecido com a atualização de técnicos. Para atualizar um utilizador, temos de ter a certeza que não alteramos a sua role, ou seja vamos forçar para que seja sempre um utilizador. É procurado e atualizado o utilizador com a role “EXT” pelo seu id e de seguida popula os testes para que estes sejam associados;

- addCovTests: encontrar e atualizar um utilizador pelo seu id e cuja role seja “EXT” e através de um push adicionar “covtest”. De seguida popular o teste.

- remCovTests: encontrar e atualizar um utilizador pelo seu id e cuja role seja “EXT” e através de um pull remove “covtest”. De seguida popular o teste.

5.5. API Middlewares

Nesta seção podemos ver os *middlewares* que foram utilizados no projeto com a sua devida explicação.

5.5.1. Authorize

```
const authorize = (opts) => {

    opts = opts || []

    return (req, res, next) => {
        if (!req.user) {
            next('Not authenticated')
        }
        const hasAuthorization = opts.includes(req.user.role)

        if (hasAuthorization) {
            next()
        } else {
            next('Not authorized')
        }
    }
}

module.exports = authorize
```

Figura 17 - Middleware authorize.js

Esta função define o método como a autenticação é feita, neste caso é através da verificação da propriedade “role” do user, caso ele tenha a “role” requerida pela função vai ter acesso a essa função caso contrário recebe a mensagem de “Not authorized”.

5.5.2. Session

```
const jwt = require('jsonwebtoken')

const {
    JWT_SECRET = 'this is for development'
} = process.env

const sessionMiddleware = (req, res, next) => {
    const sessionStr = req.cookies.session
    try {
        if (sessionStr) {
            const user = jwt.verify(sessionStr, JWT_SECRET)
            req.user = user
        } else {
            req.user = null
        }
    } catch(e) {
        // console.error(e)
        req.user = null
    }
    next()
}
```

Figura 18 - Middleware session.js

Para a sessão foi utilizada a tecnologia JSON Web Token. Esta tecnologia apresenta vantagens em relação a um início de sessão tradicional ao nível de performance.

A estratégia utilizada para validar a sessão do utilizador passa pela criação de uma cookie. Esta cookie funciona como uma chave de autenticação, quando o utilizador é validado pelo servidor. A sua validação posterior até ao término da sessão é autenticada através da cookie que foi gerada e esta cookie fica guardada no browser para que não seja necessário fazer novos pedidos ao servidor, poupando assim recursos do próprio.