Express

No último tópico foi apresentada uma revisão de aplicações web, depois uma análise do modelo de execução do JavaScript, e também uma introdução ao Node.js. No final foi abordado o desenvolvimento de uma API HTTP usando nada além do que já está disponível nativamente na plataforma.

Durante o desenvolvimento da API HTTP, algumas coisas claramente se mostraram difíceis de resolver, produzindo código frágil e mal organizado, como por exemplo:

- *Processamento do corpo da requisição:* foi necessário tratar o recebimento dos dados em "chunks". A implementação que foi feita (concatenação de strings) só funciona com dados textuais, não funcionaria para upload de arquivos, por exemplo. Se a entrada for JSON, a conversão para um objeto JavaScript e validação dos dados de entrada também foi feita manualmente, o que não é o ideal.
- *Roteamento da requisição:* foi usada uma grande carga de manipulação de strings para rotear as chamadas. Isso rapidamente deixa o código poluído e difícil de entender.
- *Tratamento de casos de erro:* como o módulo HTTP não oferece nada com relação a tratamento de erros, foi necessário codificar com cuidado para garantir que toda requisição tivesse uma resposta, mesmo em casos de exceção. Isso fica ainda pior se considerar o fato de que o processamento costuma ser assíncrono (baseado em callbacks/Promises).

Além disso, algumas coisas nem chegaram a ser abordadas, como:

- *Disponibilização de arquivos estáticos (HTML/CSS/JavaScript de front-end):* não é a única maneira, mas uma forma bem simples de disponibilizar sua aplicação para os usuários é através do próprio servidor de back-end.
- *Upload/download de arquivos*
- *Segurança:* como trabalhar a autenticação dos usuários da aplicação? Quais os mecanismos disponíveis e quando usar cada um deles?

Nesta seção será abordado como usar o framework Express para permitir o desenvolvimento de uma API com todas essas características.

Node Package Manager (NPM)

Antes de prosseguir, vale discutir sobre o gestor de pacotes do Node, o NPM. Essa ferramenta controla as dependências dos seus projetos, além de permitir a instalação de ferramentas disponíveis globalmente na instalação do Node. Para exemplificar, considere a sequência de comandos para iniciar um projeto usando Angular:

```
npm install -g @angular/cli
```

Isso instala o pacote `@angular/cli` de modo global. A partir desse comando, a ferramenta `ng` ficará disponível em qualquer terminal e em qualquer diretório. Use o site `https://www.npmjs.com` para encontrar detalhes deste e qualquer outro pacote. Antes de adicionar qualquer dependência no seu projeto, é interessante analisar a "saúde" do mesmo, que é uma análise subjetiva feita sobre a quantidade de downloads, issues abertas, data do último commit no GitHub, etc.

```
ng new
```

Esse comando inicia o projeto Angular.

```
npm install @angular/material
```

Esse comando instala a dependência `@angular/material` no projeto. Repare que o comando reclama de um `peer dependency` que você precisa instalar manualmente, o `@angular/cdk`. Uma ótima leitura sobre isso pode ser vista aqui: https://stackoverflow.com/questions/26737819/why-use-peer-dependencies-in-npm-for-plugins, e aqui: https://nodejs.org/en/blog/npm/peer-dependencies/. Também discutiremos sobre isso mais abaixo.

```
npm install @angular/cdk
```

Veja como fica o `package.json`:

```
{
  "name": "?????????????",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
  },
  "private": true,
  "dependencies": {
    "@angular/animations": "~8.0.0",
    "@angular/cdk": "^8.0.1",
    "@angular/common": "~8.0.0",
    "@angular/compiler": "~8.0.0",
    "@angular/core": "~8.0.0",
    "@angular/forms": "~8.0.0",
    "@angular/material": "^8.0.1",
    "@angular/platform-browser": "~8.0.0",
    "@angular/platform-browser-dynamic": "~8.0.0",
    "@angular/router": "~8.0.0",
    "rxjs": "~6.4.0",
    "tslib": "^1.9.0",
    "zone.js": "~0.9.1"
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~0.800.0",
    "@angular/cli": "~8.0.2",
    "@angular/compiler-cli": "~8.0.0",
    "@angular/language-service": "~8.0.0",
    "@types/node": "~8.9.4",
    "@types/jasmine": "~3.3.8",
    "@types/jasminewd2": "~2.0.3",
    "codelyzer": "^5.0.0",
    "jasmine-core": "~3.4.0",
    "jasmine-spec-reporter": "~4.2.1",
    "karma": "~4.1.0",
```

```
    "karma-chrome-launcher": "~2.2.0",
    "karma-coverage-istanbul-reporter": "~2.0.1",
    "karma-jasmine": "~2.0.1",
    "karma-jasmine-html-reporter": "^1.4.0",
    "protractor": "~5.4.0",
    "ts-node": "~7.0.0",
    "tslint": "~5.15.0",
    "typescript": "~3.4.3"
  }
}
```

O que são os `scripts`? São utilitários que você deseja deixar documentado no próprio projeto, acessíveis através do comando `npm run <script>`, ex: `npm run start`. Alguns desses scripts podem ser chamados sem o uso do comando `run`, como `start`: `npm start`.

As `dependencies` são todas as bibliotecas usadas diretamente pelo seu projeto. Elas são instaladas no diretório `node_modules` (esse diretório *não* deve ser versionado no repositório do projeto). Sempre que uma cópia limpa do projeto for criada, basta executar o comando `npm install` para baixar novamente todas as dependências.

As `devDependencies` são normalmente ferramentas usadas no ambiente de desenvolvimento, mas que não são necessárias para usar uma biblioteca ou rodar o projeto em um ambiente.

Execute o comando abaixo para analisar o gráfico de dependências do projeto:

```
npm ls
```

Sim, praticamente a Internet toda. Para ter uma saída um pouco mais legível, limite o nível de folhas:

```
npm ls --depth=0
```

Semantic Versioning (SemVer)

Dê uma olhada nas dependências do projeto usado para testes na seção anterior. O que significam, exatamente, as definições de versão nas dependências? Por exemplo:

```
"@angular/cli": "~8.0.2"
```

Todo pacote Node que é publicado em um repositório (como o npmjs.com, por exemplo) precisa ter um nome e uma versão. Essa versão precisa ser compatível com o conceito de versão semântica (SemVer [1][2]). De forma resumida, versionamento semântico é composto por três números:

```
1.2.3
```

O primeiro é a versão `MAJOR`. O segundo, a `MINOR`, e o terceiro, `PATCH`. A versão `MAJOR` só deve mudar quando a biblioteca alterar sua *interface externa* de modo *incompatível* com as versões anteriores, por exemplo: alteração de assinatura de método de modo que chamadas existentes deixem de funcionar, remoção completa de um método. A `MINOR` é incrementada quando houver *adição de funcionalidade*, sem quebrar código existente. Por fim, a versão `PATCH` só é incrementada quando existirem apenas correções de bugs, melhorias de performance e coisas do tipo em funcionalidades existentes.

Com isso, é possível dizer para o NPM quais versões de determinada biblioteca você entende que seu projeto suporta. A maneira mais engessada seria dizer *exatamente* qual versão usar, por exemplo:

```
"@angular/cli": "8.0.2"
```

Neste caso, não importa quantas novas versões da biblioteca já foram publicadas, esse projeto sempre vai usar a versão `8.0.2`. Uma outra forma é permitir que o npm baixe versões `PATCH` mais novas automaticamente. Isso pode ser atingido de diversas formas:

```
"@angular/cli": "8.0"
"@angular/cli": "8.0.x"
"@angular/cli": "8.0.X"
"@angular/cli": "8.0.*"
"@angular/cli": "~8.0.2"
"@angular/cli": ">=8.0.2 <8.1"
"@angular/cli": "8.0.2 - 8.0"
```

Note que as 4 primeiras opções vão aceitar a versão `8.0.1` se esta estiver disponível, enquanto as 3 últimas são um pouco mais restritivas.

De modo similar, você pode ser ainda mais flexível, e aceitar incrementos `MINOR` de forma automática:

```
"@angular/cli": "~8"
"@angular/cli": "8.x"
"@angular/cli": "8"
"@angular/cli": ">=8.0.2 <9"
"@angular/cli": ">=8.0.2 - 8"
"@angular/cli": "^8.0.2"
```

E se estiver se sentindo aventureiro, pode até mesmo aceitar qualquer versão da biblioteca:

```
"@angular/cli": "*"
```

[1] https://semver.org/

[2] https://docs.npmjs.com/misc/semver

Instalação do Express e primeiro projeto

Agora que ficou claro o uso do `npm`, é possível iniciar um novo projeto e adicionar o framework `express` como dependência. Vá até um diretório vazio na sua máquina e digite o seguinte comando:

```
npm init
```

Se o seu projeto não vai ser consumido por outros como uma biblioteca, é uma boa ideia remover o atributo `main` e adicionar `private: true` no arquivo `package.json` que foi gerado.

Adicione agora a dependência para a versão mais recente do Express:

```
npm install express
```

O que é esse arquivo `package-lock.json` que foi criado? Está fora do escopo desta disciplina, mas duas ótimas discussões sobre este arquivo podem ser encontradas aqui: https://renovatebot.com/docs/dependency-pinning/ e aqui: https://github.com/commitizen/cz-conventional-changelog-default-export/pull/4#issuecomment-358038966.

Agora crie um arquivo chamado `index.js`, colocando o conteúdo abaixo:

```
const express = require('express');
const app = express()


app.get('/ola', (req, res) => {
    const nome = req.query.nome;
    res.send(`Olá, ${nome}!`);
});

app.listen(3000, () => {
    console.log('Primeira API com Express.');
});
```

Note que já é possível observar algumas diferenças drásticas com relação ao tópico anterior. Primeiro, não foi necessário fechar a resposta, o próprio método `send` já faz isso. Segundo, o objeto `req` vem enriquecido com dados da requisição, não há a necessidade de processar esses dados na mão. Terceiro, o roteamento fica muito mais simples, por padrão a resposta sempre será 404, a não ser que o caminho e método se mostre compatível a alguma das chamadas no estilo `app.get`. Veja esse outro exemplo, envolvendo parâmetros de caminho e corpo:

```
app.use(express.json());

app.post('/pessoas/:id/telefones', (req, res) => {
    const idPessoa = req.params.id;
    const telefone = req.body;
    console.log(idPessoa);
    console.log(telefone);
    res.send();
});
```

Note que o atributo `body` só ficou disponível depois de instalarmos um `middleware` de processamento de corpo de requisição.

Express Generator

Existe uma ferramenta de geração de código disponível para usuários de Express, chamada Express Generator. Para utilizá-la, primeiro instale-a usando `npm`:

```
npm install -g express-generator
```

Agora, em um diretório, execute o seguinte comando para criar a estrutura básica de um projeto (será criado um diretório com o nome `02_generator`):

```
express 02_generator
```

Entre no diretório do projeto e instale as dependências:

```
npm install
```

E execute-o com o comando `npm start`. Você pode acessar a aplicação no navegador, no endereço `http://localhost:3000`.

Algumas observações:

- Para uma API HTTP, você provavelmente não vai usar a parte de `views`. É possível gerar um projeto sem essa parte, com o comando `express --no-view`.
- Repare que o diretório `public` é servido como um diretório de arquivos estáticos, isso resolve mais um dos pontos propostos no início deste tópico. A "mágica" ocorre no arquivo `app.js`, na linha que instala o *middleware* `express.static` na aplicação. Mais detalhes sobre middlewares serão fornecidos mais abaixo.
- É possível observar aqui que o Express estimula a modularização dos endpoints em `Routers`. Uma boa analogia é considerar cada `Router` como uma mini aplicação express, *embutida* na aplicação raíz.

A decisão entre usar o Express Generator ou não cabe a cada equipe, mas normalmente o ideal é começar do zero, adicionando apenas aquilo que o projeto de fato necessita. O valor desse tipo de ferramenta está na fase de aprendizado, para aqueles que querem ter algo funcionando para então ter uma ideia daquilo que é considerado boa prática.

Middlewares

Antes de refazer o CRUD de tarefas usando Express, vale discutir sobre o conceito de `Middlewares`. Um Middleware, dentro do Express, nada mais é uma função que será chamada para todas as requisições, capaz de enriquecer os dados da requisição/resposta, finalizar uma resposta automaticamente (tratamento de erros, segurança, etc), dentre outras. Uma forma de ver é comparar com o conceito de `plug-in`. Veja o exemplo abaixo, que demonstra o funcionamento dos middlewares na implementação de auditoria e segurança (bem rudimentares):

```
const express = require('express');
const url = require('url');
const app = express();

app.use((req, res, next) => {
    console.log('Interceptando a requisição: ', req.url);
    const reqUrl = url.parse(req.url, true);
    if (reqUrl.path.startsWith('/admin')) {
        const senha = req.header('X-SenhaAdmin');
        if (senha === 'senha123') {
            next();
        } else {
            res.status(403).send('Apenas administradores podem acessar este recurso');
        }
    } else {
        next();
    }
});

app.get('/clientes', (req, res) => res.send(['João', 'Maria']));

app.get('/admin/clientes/restritos', (req, res) => res.send(['João']));

app.listen(3000);
```

Note que esse não é nem de longe um código pronto para ser usado em um projeto real, tente acessar `http://localhost:3000/Admin/clientes/restritos` (com o A maiúsculo) por exemplo e veja o motivo. Questões críticas como segurança devem ser implementadas com muito cuidado, e preferencialmente delegadas para bibliotecas/frameworks especializados nisso. No entanto, como o objetivo aqui é exercitar o desenvolvimento de middlewares customizados, a fragilidade não tem tanta importância.

Por mais que seja possível e até incentivado o desenvolvimento de middlewares customizados, o mais comum é que pacotes de funcionalidades externos sejam adicionados dessa forma, como é o caso do middleware `express.static` que vimos na seção anterior.

Refazendo o CRUD de tarefas

Para continuar demonstrando o uso do Express, vale refazer alguns endpoints do tópico anterior. Como ainda não será utilizado Knex e uma base relacional, o módulo `tarefas/tarefas-modelo.js` pode ser copiado do jeito que estava, para este novo projeto.

Comece criando um novo projeto Express:

```
npm init
npm i express
```

E criando um arquivo `index.js` com o seguinte conteúdo:

```
const express = require('express');
const app = express();

app.use(express.json());

app.listen(3000);
```

Por mais que seja possível implementar todos os endpoints direto neste arquivo, conforme o projeto cresce essa prática fica completamente inviável. A solução do Express para este problema de modularização é a definição de `Routers`. Crie uma pasta chamada `tarefas`, com um arquivo chamado `tarefas-router.js` dentro dela, com o seguinte conteúdo:

```
const express = require('express');
const router = express.Router();

const modelo = require('./tarefas-modelo');


router.get('/', (req, res) => {
    res.send(modelo.listar(req.query.filtro));
});

router.get('/:id', (req, res) => {
    res.send(modelo.buscarPorId(req.params.id));
});

router.post('/', (req, res) => {
    const body = req.body;
    const tarefa = new modelo.Tarefa(body.descricao, body.previsao);
```

```
    modelo.cadastrar(tarefa);
    res.send();
});

module.exports = router;
```

Copie o `tarefas-modelo.js` desenvolvido na seção anterior para a pasta `tarefas`.

Por fim, ajuste o `index.js` para configurar o router como se ele fosse um middleware (de fato todo router é um middleware de uma aplicação Express):

```
const express = require('express');
const app = express();

const tarefasRouter = require('./tarefas/tarefas-router');


app.use(express.json());
app.use('/tarefas', tarefasRouter);

app.listen(3000);
```

Tire um tempo para comparar essa solução com a do tópico anterior, especialmente as partes que ficaram abstraídas pelo Express.

Validação de dados de entrada

Uma das características discutidas no início deste tópico foi a validação de dados de entrada. O framework Express não possui nada nativo com relação a isso (exceto a conversão automática de `json`, que já vem sendo usada nas seções anteriores), mas a comunidade criou várias bibliotecas que permitem adicionar esse tipo de validação no seu projeto. Uma delas é a `express-validator`. Existem várias maneiras de usá-la, e neste tópico serão apresentadas duas delas. Primeiro instale a ferramenta no projeto:

```
npm i express-validator
```

A primeira forma de usá-la é adicionando uma lista de validações como middlewares em cada endpoint. Veja:

```
const express = require('express');
const { check, validationResult } = require('express-validator');
const router = express.Router();

const modelo = require('./tarefas-modelo');


router.get('/', (req, res) => {
    res.send(modelo.listar(req.query.filtro));
});

router.get('/:id', (req, res) => {
    res.send(modelo.buscarPorId(req.params.id));
});
```

```
router.post('/',
    check('descricao').isLength({ min: 3, max: 100 })
        .withMessage('Deve ser um valor entre 3 e 100 caracteres.'),
    check('previsao').toDate().not().isEmpty()
        .withMessage('Deve ser uma data válida.'),
    (req, res) => {

        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            res.status(400).json({ errors: errors.array() });
            return;
        }

        const body = req.body;
        const previsao = body.previsao.toISOString();
        const tarefa = new modelo.Tarefa(body.descricao, previsao);
        modelo.cadastrar(tarefa);
        res.send();
    });

module.exports = router;
```

A parte importante aqui é a inclusão da dependência `express-validator`, o uso de cadeias de validação (chamadas `check` no handler de cadastro de tarefa) e o uso do método `validationResult` para verificar se houve erro de validação ou não, antes de prosseguir com o tratamento normal da requisição.

A outra maneira é através do uso do middleware `checkSchema`:

```
const express = require('express');
const { checkSchema, validationResult } = require('express-validator');
const router = express.Router();

const modelo = require('./tarefas-modelo');


router.get('/', (req, res) => {
    res.send(modelo.listar(req.query.filtro));
});

router.get('/:id', (req, res) => {
    res.send(modelo.buscarPorId(req.params.id));
});

router.post('/',
    checkSchema({
        descricao: {
            in: 'body',
            errorMessage: 'Deve ser um valor entre 3 e 100 caracteres.',
            isLength: {
                options: {
                    min: 3,
```

```
                    max: 100
                }
            }
        },
        previsao: {
            in: 'body',
            errorMessage: 'Deve ser uma data válida.',
            toDate: true,
            isEmpty: {
                negated: true
            }
        }
    }),
    (req, res) => {

        const errors = validationResult(req);
        if (!errors.isEmpty()) {
            res.status(400).json({ errors: errors.array() });
            return;
        }

        const body = req.body;
        const previsao = body.previsao.toISOString();
        const tarefa = new modelo.Tarefa(body.descricao, previsao);
        modelo.cadastrar(tarefa);
        res.send();
    });

module.exports = router;
```

As duas formas possuem prós e contras, cabe à equipe do projeto decidir qual ou quais usar no código-fonte.

Autenticação e autorização

Boa parte das aplicações não triviais possui a necessidade de autenticar seus usuários, e autorizar o acesso a funcionalidades e dados. No caso do CRUD de tarefas, pode-se pensar em autenticação como um formulário que solicita usuário e senha, e autorização como permitir apenas que o usuário visualize e altere suas próprias tarefas. Um exemplo mais complicado de autenticação seria por exemplo integrar com o Google Accounts, e um exemplo de autorização seria permitir que um usuário compartilhasse tarefas com outros, que passariam a ter acesso àqueles dados e talvez até a modificá-los.

Existem muitas técnicas para implementar autorização em chamadas de API, mas normalmente elas giram em torno de passar no header de requisição `Authorization` um valor, seja ele qual for, capaz de provar para o servidor que quem submeteu aquela requisição é quem diz ser.

Uma delas é a chamada autorização `Basic`, onde o usuário e senha são passados em cada requisição feita para o servidor. Para implementar essa autorização, o primeiro passo é responder com status code `401` as requisições que não possuírem nenhum valor no header `Authorization`:

```
const express = require('express');
const app = express();
```

```
const tarefasRouter = require('./tarefas/tarefas-router');

app.use((req, res, next) => {
    const auth = req.header('Authorization');
    if (!auth || !auth.startsWith('Basic ')) {
        res.status(401)
            .header('WWW-Authenticate', 'Basic realm="tarefas"')
            .send();
    } else {
        next();
    }
});

app.use(express.json());
app.use('/tarefas', tarefasRouter);

app.listen(3000);
```

Note que, ao abrir uma URL no navegador, ele vai mostrar um diálogo solicitando usuário e senha.

O próximo passo é de fato checar as credenciais contra qualquer que seja a base de dados de usuários:

```
const express = require('express');
const app = express();

const tarefasRouter = require('./tarefas/tarefas-router');

app.use((req, res, next) => {
    const auth = req.header('Authorization');
    if (!auth || !auth.startsWith('Basic ')) {
        res.status(401)
            .header('WWW-Authenticate', 'Basic realm="tarefas"')
            .send();
    } else {
        const token = auth.substr('Basic '.length);
        const [ user, pass ] = Buffer.from(token, 'base64').toString('UTF-
8').split(':');
        if (user == 'samuel' && pass == '123') {
            next();
        } else {
            res.status(403).send();
        }
    }
});

app.use(express.json());
app.use('/tarefas', tarefasRouter);

app.listen(3000);
```

Ao invés de implementar tudo isso na mão (e acabar com código confuso responsável por fazer conversão de base64 e manipulação de strings), mais uma vez é possível usar bibliotecas de middleware contribuídas de forma aberta pela comunidade. Adicione no projeto a dependência `express-basic-auth`:

```
npm i express-basic-auth
```

E configure o middleware na aplicação:

```
const express = require('express');
const basicAuth = require('express-basic-auth');
const app = express();

const tarefasRouter = require('./tarefas/tarefas-router');

app.use(basicAuth({
    users: {
        'samuel': '123',
        'admin': '234'
    },
    realm: 'tarefas',
    challenge: true
}));

app.use((req, res, next) => {
    console.log(req.auth);
    next();
});

app.use(express.json());
app.use('/tarefas', tarefasRouter);

app.listen(3000);
```

Repare que, como demonstrado no middleware customizado, o `express-basic-auth` disponibiliza os dados do usuário autenticado no atributo `auth` da requisição, permitindo por exemplo que apenas as tarefas daquele usuário sejam retornadas.

Em um projeto real, dificilmente será utilizada a autenticação básica entre front-end e back-end. Ela serve mais como um legado e ainda é usada em algumas integrações entre back-ends diferentes. Uma outra opção é o uso de tokens, gerados pelo próprio back-end (normalmente em uma função de autenticação, com acesso livre) e capazes de serem validados. Implementações mais antigas seguiam uma lógica mais ou menos parecida com a que segue:

1. Usuário não autenticado era encaminhado para formulário de login;
2. Usuário submete login e senha;
3. Servidor valida o usuário e senha, gerando um registro no banco de dados com um *token* aleatório (por exemplo um GUID);
4. Navegador do usuário armazena esse token e envia no header `Authorization` em cada futura chamada que fizer.
5. Servidor valida no banco esse token a cada requisição.

Esta estrutura atende, mas além de ser um pouco complexa de implementar, não escala muito bem, visto que toda requisição acaba tendo que validar o token no banco de dados. Uma solução um pouco

melhor envolve o uso de JSON Web Tokens. A ideia é que não haja mais a necessidade de armazenar os tokens no banco de dados. O funcionamento detalhado por trás dos JWT está fora do escopo dessa disciplina, e pode ser encontrado aqui: https://jwt.io/.

Para adicionar autenticação via JWT no projeto, instale a dependência `express-jwt`:

```
npm i express-jwt
```

Depois configure o middleware, tomando nota do `secret` utilizado (ele deverá ser o mesmo na hora de gerar o token):

```
const express = require('express');
const jwt = require('express-jwt');
const app = express();

const tarefasRouter = require('./tarefas/tarefas-router');

app.use(jwt({
    secret: 'tarefasjwtsecret'
}));

app.use((req, res, next) => {
    console.log(req.user);
    next();
});

app.use(express.json());
app.use('/tarefas', tarefasRouter);

app.listen(3000);
```

Note que, diferentemente do `express-basic-auth`, o middleware `express-jwt` disponibiliza os dados do usuário autenticado no atributo `req.user`. Para testar, gere um token no site `https://jwt.io/`, atentando-se para usar o mesmo segredo. Não há formato específico para os `claims` do token, mas é boa prática usar o `sub` (subject, sujeito) para representar o login e `name` para representar o nome do usuário. Também é boa prática definir uma data de expiração na claim `exp`.

Mas como gerar esse token dentro do próprio back-end? Para isso você precisa de uma biblioteca de geração de tokens. Um exemplo para o ecossistema Node.js é a biblioteca `jsonwebtoken`. Instale-a:

```
npm i jsonwebtoken
```

Agora adicione um endpoint para processar as requisições de login:

```
const express = require('express');
const jwt = require('express-jwt');
const jsonwebtoken = require('jsonwebtoken');
const app = express();

const tarefasRouter = require('./tarefas/tarefas-router');

app.use(jwt({
    secret: 'tarefasjwtsecret'
```

```
}).unless({ path: '/login' }));

app.use((req, res, next) => {
    console.log(req.user);
    next();
});

app.use(express.json());

app.post('/login', (req, res) => {
    const { user, pass } = req.body;
    if (!user || !pass) {
        res.status(400).send('Informe o usuário e a senha.');
    } else if (user !== 'samuel' || pass !== '123') {
        res.status(400).send('Credenciais inválidas.');
    } else {
        const token = jsonwebtoken.sign({
            sub: user,
            exp: new Date(2020, 1, 1, 10, 30, 0).getTime() / 1000
        }, 'tarefasjwtsecret');
        res.send(token);
    }
});

app.use('/tarefas', tarefasRouter);

app.listen(3000);
```

Note que o middleware `jwt` teve que ser alterado ligeiramente para ignorar requisições feitas no caminho `/login`.

Para testar, faça uma requisição POST para o caminho `/login`, passando como corpo um objeto com os atributos `user` e `pass`. Depois, com o JWT recebido na resposta, valide-o no site `https://jwt.io/` e use-o para autenticar chamadas nos endpoints de tarefas. Repare que se você usar uma data já passada no campo `exp` do token, o middleware vai rejeitar a autenticação.

Trabalhando com uploads

Não é raro uma API precisar trabalhar com upload de documentos e imagens. Neste tópico será apresentada uma abordagem de implementação, juntamente com um front-end estático. De forma geral, essa aplicação irá:

1. Mostrar um campo para seleção de imagem.
2. Ao selecionar uma imagem, ela será apresentada para que o usuário selecione uma parte dela (crop) usando uma biblioteca JavaScript.
3. Ao confirmar, a imagem será enviada para o servidor.

Para começar, crie uma nova aplicação e adicione o middleware `express.static`:

```
const express = require('express');
const app = express();

app.use(express.static('public'));
app.use(express.json());
```

```
app.listen(3000);
```

Crie agora um arquivo `public/index.html` com o seguinte conteúdo:

```
<!DOCTYPE html>
<html>
    <body>
        <p>
            Selecione a imagem desejada:
            <input type="file">
        </p>
    </body>
</html>
```

Antes de enviar a imagem para o back-end, permita que o usuário selecione uma parte dela, uma operação de *cropping*. Como isso não é nativo do navegador, use uma biblioteca JavaScript, como por exemplo a `cropperjs` [1]. Comece adicionando a biblioteca no `index.html` :

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/cropperjs/1.5.1/cropper.css" />
    </head>
    <body>
        <p>
            Selecione a imagem desejada:
            <input type="file" onchange="onFileChange()">
        </p>
        <script
src="https://cdnjs.cloudflare.com/ajax/libs/cropperjs/1.5.1/cropper.js"></script>
    </body>
</html>
```

Note que essas dependências estão sendo adicionadas via CDN e não via npm, pois neste caso não estamos usando o `npm` no parte front-end do projeto. Se fosse um projeto Angular, por exemplo, a dependência poderia ser adicionada via `npm i cropperjs` .

O próximo passo é capturar o evento de seleção de imagem, mostrá-la em uma tag `img` e inicializar o CropperJS nela:

```
<!DOCTYPE html>
<html>
    <head>
        <link rel="stylesheet"
href="https://cdnjs.cloudflare.com/ajax/libs/cropperjs/1.5.1/cropper.css" />
    </head>
    <body>
        <p id="form">
            Selecione a imagem desejada:
            <input type="file" onchange="onFileChange(event)">
        </p>
```

```
        <script
src="https://cdnjs.cloudflare.com/ajax/libs/cropperjs/1.5.1/cropper.js"></script>
        <script>

            function onFileChange(event) {
                const file = event.target.files[0];

                const reader = new FileReader();
                reader.onload = () => {
                    const el = document.getElementById('form');

                    const img = document.createElement('img');
                    img.style.width = '250px';
                    img.src = reader.result;

                    el.innerHTML = '';
                    el.appendChild(img);

                    const cropper = new Cropper(img, {});

                    const button = document.createElement('button');
                    button.onclick = () => {
                        console.log(cropper.getData());
                    };
                    button.innerText = 'Confirmar';
                    el.append(button);

                };
                reader.readAsDataURL(file);

            }

        </script>
    </body>
</html>
```

Note que no evento do clique do botão é possível obter as coordenadas e o tamanho do quadro de crop selecionado pelo usuário.

O próximo passo é criar um endpoint no back-end capaz de receber o arquivo, os dados de crop e armazenar a imagem final. O Express não dá suporte para uploads de modo nativo, mas existe um ótimo middleware que fornece essa funcionalidade, chamado `express-fileupload`. Instale-o usando `npm`:

```
npm i express-fileupload
```

Configure-o no arquivo `index.js`:

```
const express = require('express');
const fileUpload = require('express-fileupload');
const app = express();

app.use(express.static('public'));
```

```
app.use(express.json());

app.use(fileUpload({
    limits: {
        fileSize: 50 * 1024 * 1024 // 50mb
    }
}));

app.post('/upload', (req, res) => {
    const imagem = req.files.imagem;
    const { cropx, cropy, cropwidth, cropheight } = req.body;
    console.log(imagem);
    console.log(cropx, cropy, cropheight, cropwidth);
    res.send();
});

app.listen(3000);
```

Note que foi criado também um endpoint respondendo requisições POST no caminho `/upload`, para permitir testes. Altere agora o arquivo `index.html` para chamar esse endpoint:

```
button.onclick = () => {

    const data = new FormData();
    data.append('imagem', file);

    const crop = cropper.getData();
    data.append('cropx', crop.x);
    data.append('cropy', crop.y);
    data.append('cropwidth', crop.width);
    data.append('cropheight', crop.height);

    fetch('/upload', {
        method: 'POST',
        body: data
    }).then(resp => {
        if (resp.status == 200) {
            el.innerText = 'Sucesso!';
        } else {
            el.innerText = 'Erro!';
        }
    }).catch(err => {
        console.error(err);
        el.innerText = 'Erro!';
    });

};
```

A parte importante aqui é a preparação do objeto `data` com o `file` (obtido à partir do elemento `input`) e os dados de crop escolhidos pelo usuário.

Manipulação de imagens também não está disponível nativamente, e uma ótima opção é a biblioteca `sharp` [2]. Instale-a usando npm:

```
npm i sharp
```

Ajuste agora o código do endpoint para efetuar a operação de crop e armazenar a imagem resultante no sistema de arquivos:

```
app.post('/upload', (req, res) => {
    const imagem = req.files.imagem;
    const { cropx, cropy, cropwidth, cropheight } = req.body;

    sharp(imagem.data)
        .extract({
            top: parseInt(cropy),
            left: parseInt(cropx),
            height: parseInt(cropheight),
            width: parseInt(cropwidth)
        })
        .toFile('resultado.png')
        .then(() => res.send())
        .catch(err => {
            console.error(err);
            res.status(500).send();
        });

});
```

Pronto, com isso você concluiu o endpoint e o front-end capazes de armazenar uma parte de uma imagem, conforme orientações do próprio usuário.

[1] https://github.com/fengyuanchen/cropperjs/blob/master/README.md#features

[2] https://github.com/lovell/sharp