Você será capaz de:

- Entender o que há por dentro de um token de autenticação e autorização;
- Gerar tokens a partir de informações como login e senha;
- Autenticar pessoas usuárias utilizando o token JWT.
- Autorizar o acesso a rotas do Express, usando o token JWT.

Por que isso é importante?

O JWT é, definitivamente, uma maneira inteligente de obter, com segurança, a identidade de um usuário!
Imagine que você tem uma aplicação em que você precisa verificar se a sessão de uma pessoa ainda está ativa, mesmo depois de ela ter desligado o computador/smartphone. E aí, como faz?
Você poderia usar cookies. Porém, usar cookies, atualmente, não é uma boa opção por várias razões: o usuário pode não aceitar seus cookies, você não tem controle de onde ele está rodando, o site fica mais pesado dependendo de qual cookie você está usando, etc.
Uma alternativa é usar o `JWT` , que te disponibiliza um token/hash/código criptografado que você pode enviar para uma API e validá-lo como preferir.
Além disso, essa tecnologia nos traz outros benefícios:

1. Não utiliza banco de dados: usar o JWT implica menos consultas ao banco de dados, o que nos dá um tempo de resposta mais rápido. Caso você esteja usando serviços pagos, como o DynamoDB , que cobram por consulta, o JWT poderá reduzir os custos de consumo.
2. Mais simples de usar (se você for cuidadoso): se seu projeto não tem uma arquitetura boa para administrar as sessões dos seus clientes, e seus princípios básicos de segurança não forem claros, o tempo de desenvolvimento usando JWT será bem mais rápido, considerando que você pode simplesmente usar alguma biblioteca existente.
3. Utilizado em vários serviços: você pode ter um servidor de autenticação que lida com o login/cadastro para gerar o token para a pessoa usuária. A partir daí, você pode transitar seu token entre várias aplicações, sendo necessário logar apenas uma vez e logo depois estar logado em todas as outras aplicações do seu sistema.

> No Google, por exemplo, você loga uma única vez e pode usar serviços como Google Drive, Gmail, Google docs, Google fotos, etc. sem precisar logar novamente.

O que é o JWT?

Vamos começar assistindo a um vídeo:

VIDEO: 28.1 JWT 08:18

Interessante, né? O JWT (JSON Web Token) é um token gerado a partir de dados "pessoais" que pode ser trafegado pela internet ao fazer requisições para APIs e afins. Mas atenção: toda a informação que colocamos no JWT é pública , e qualquer pessoa com o token consegue ler essas informações. O mecanismo de segurança do JWT permite, no entanto, que apenas quem tem a senha consiga alterar as informações contidas em um token.
A coisa toda funciona assim:

- O navegador solicita que o usuário digite seu login e senha.
- O navegador então envia esse login e senha ao servidor, para verificar que esses dados estão corretos.
- Uma vez que valida login e senha, o servidor cria dois objetos: um contendo informações sobre o token que será gerado, que chamamos de header , e outro contendo os dados do usuário e as permissões que aquela pessoa têm, ao qual chamamos de payload .
- O servidor então converte esses dois objetos em JSON, junta-os em uma mesma string e utiliza um algoritmo chamado HMAC para "criptografar" essa string usando um "segredo" que só ele sabe, gerando o que chamamos de assinatura , que nada mais é do que Header + Payload criptografados.
- Por fim, o servidor combina o header, o payload originais e a assinatura, criando assim o token .
- O token é enviado ao cliente, que o armazena para utilizá-lo nas próximas requisições.

Chamamos de autenticação o processo pelo qual a pessoa usuária consegue, utilizando informações confidenciais como email e senha, efetuar login com sucesso em uma aplicação, tendo como retorno um

JSON Web Token pelo qual é possível acessar suas permissões de navegação.
Na próxima requisição...

- O navegador envia ao servidor os dados para, por exemplo, cadastrar um novo produto. Juntamente a esses dados, o navegador envia o token que recebeu ao realizar o login.
- Quando recebe os dados, a primeira coisa que o servidor faz é obter o Header e o Payload do token e criptografá-los, gerando, mais uma vez, a assinatura.
- O servidor, então, compara a nova assinatura com a assinatura que foi enviada pelo client. Se ambas forem iguais, quer dizer que o conteúdo do Header e do Payload não foram modificados desde o login.
- Agora que já sabe que o token é válido, o servidor continua processando a requisição e, por fim, entrega a resposta para o cliente.

O JWT também é usado para autorização, quando precisamos fazer o processo de atestar as permissões de uma pessoa usuária que deseja acessar uma rota ou recurso protegido. Isso exige o envio do token, normalmente no header Authorization, a partir do qual são acessadas as informações necessárias para a verificação.
Mas o que acontece se, antes de tentar cadastrar um produto, a pessoa que está usando nossa aplicação tentar alterar o token?
Suponha que o payload do token possui uma propriedade chamada `admin` e que, no token da pessoa em questão, possui o valor `false` . A pessoa, a fim de tentar obter privilégios de administradora indevidamente, altera o payload, setando o valor de `admin` para `true` . Ela então armazena esse token modificado na aplicação e tenta cadastrar um produto. Nesse caso, o que acontece do lado do servidor?
Antes de continuar a leitura, tente descrever o processo que acontece do lado do servidor
Acontece o seguinte:

- O cliente envia, para o servidor as informações do produto e o token modificado
- O servidor extrai o payload e header do token e, utilizando essas duas informações, gera uma assinatura.
- Ao comparar a assinatura nova com a assinatura enviada pelo client, o servidor percebe que há uma diferença! Isso acontece porque criptografar `{ "admin": false }` sempre vai gerar um resultado

(uma assinatura, nesse caso) diferente de criptografar `{ "admin": true }`.
- Como a assinatura é diferente, o servidor rejeita a requisição, devolvendo um status de erro com uma mensagem informando que o token é inválido.

Perceba que, para que a pessoa usuária consiga alterar o seu token e obter privilégios a mais, ela precisaria gerar uma nova assinatura. Acontece que, para gerar uma nova assinatura, é necessário possuir o segredo, que apenas o servidor possui. Sendo assim, é virtualmente impossível adulterar um token JWT, o que torna essa tecnologia muito confiável para tratar de autenticação.

Autenticação e Autorização

É importante ressaltar que autenticação e autorização são coisas diferentes. Autenticação é usada para atestar que alguém é quem diz ser, verificando sua identidade, comumente feita por meio de informações confidenciais como email e senha. Já a autorização verifica as permissões de uma pessoa para acessar ou executar determinadas operações.
Um exemplo simples que evidencia essa diferença é quando você faz *log in* em uma rede social. Depois de atestar que o nome e senha conferem, você está devidamente autenticado e pode navegar pela aplicação e fazer diversas operações. Mas ao tentar, por exemplo, apagar uma foto de outra pessoa, você provavelmente não terá êxito, uma vez que geralmente, cada cliente só tem autorização para apagar suas próprias postagens.
O simples fato de se estar autenticado pode dar várias permissões para a pessoa usuária, mas ainda pode haver situações em que sejam exigidas autorizações extras, além da autenticação inicial. A partir disso, podemos concluir que a autenticação sempre precede a autorização.
Abaixo, vamos explicar mais sobre o HMAC e a criptografia envolvida no processo, mas não se assuste: não vamos implementar nada disso na mão; tudo está encapsulado nas bibliotecas do JWT.

O que é HMAC?

O HMAC é um algoritmo para gerar um `MAC` (código de autenticação de mensagem) criptografado através de algum algoritmo de hash (algoritmos que codificam mensagens), como `md5`, `sha1` ou `sha256`, a partir de uma chave secreta (uma senha) e de uma mensagem qualquer. Por exemplo, se gerarmos o HMAC da mensagem "Olá, tudo bem?", com o

segredo "minhaSenha" e o algoritmo `sha1` , teremos o seguinte resultado: `b88651e71c7c757560722b52e5f1ead754a759d8` . No entanto, se alterarmos o texto para "olá, tudo bem?", mudando apenas a capitalização da primeira letra, o resultado passa a ser `ac7016fd2014ca9a79ac2e3ef16b6bd857f91f7a` . Agora, imagine que, ao invés de "Olá, tudo bem?" façamos isso com o payload do nosso token. Ao mudar qualquer mínimo detalhe das informações daquele token, a assinatura se torna inválida.
Curiosidade: A fórmula do HMAC é a seguinte:
`HMAC(K, m) = hash(K1 + hash(K2 + m))`
onde:

- `K` é a chave secreta;
- `m` é a mensagem;
- `hash` é a função de hash escolhida (md5, sha1 etc);
- `K1` e `K2` são chaves secretas derivadas da chave original K;
- `+` é a operação de concatenação de strings.

Entendendo o JWT

O resultado final do JWT dá-se através da assinatura criptográfica de dois blocos de JSON codificados em base64. Esses dois blocos JSON codificados são o *header* (cabeçalho) e *payload* (carga) que mencionamos acima. A *signature* (assinatura) é a junção dos hashes gerados a partir do header e payload.
Header
O Header contém duas propriedades: o tipo do token, que é JWT, e o tipo do algoritmo de *hash* , como `HMAC-SHA256` ou `RSA` :

Copiar

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Payload (dados do usuário)
Esta é a segunda parte do token, que contém os "dados". Esses "dados" são declarações sobre uma entidade (geralmente, o usuário):

Copiar

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
```

```
}
```

Signature
Para gerar a assinatura, você deve usar o header e o payload codificados em `base64` , usando o algoritmo definido no header:
Código de exemplo:

Copiar

```
import { hmac } from 'bibliotecaDeHmac';

function base64 (string) {
  return Buffer.from(string).toString('base64');
}

const header = JSON.stringify({
  alg: 'HS256',
  type: 'JWT'});

const payload = JSON.stringify({
  sub: '1234567890',
  name: 'John Doe',
  admin: true});

const secret = 'MinhaSenhaMuitoComplexa123';

const assinatura = hmac(`${base64(header)}.${base64(payload)}`,
secret);

const token =
`${base64(header)}.${base64(payload)}.${base64(assinatura)}`;
```

PS: você verá código de verdade daqui a pouco :)
O resultado terá a seguinte estrutura:
`(Header em base64).(Payload em base64).(Signature em base64)`
Exemplo de resultado:
`eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InVzZXIxIiwiZXhwIjoxNTQ3OTc0MDgyfQ.2Ye5_w1z3zpD4dSGdRp3s98ZipCNQqmsHRB9vioOx54`
Nesse caso, temos:

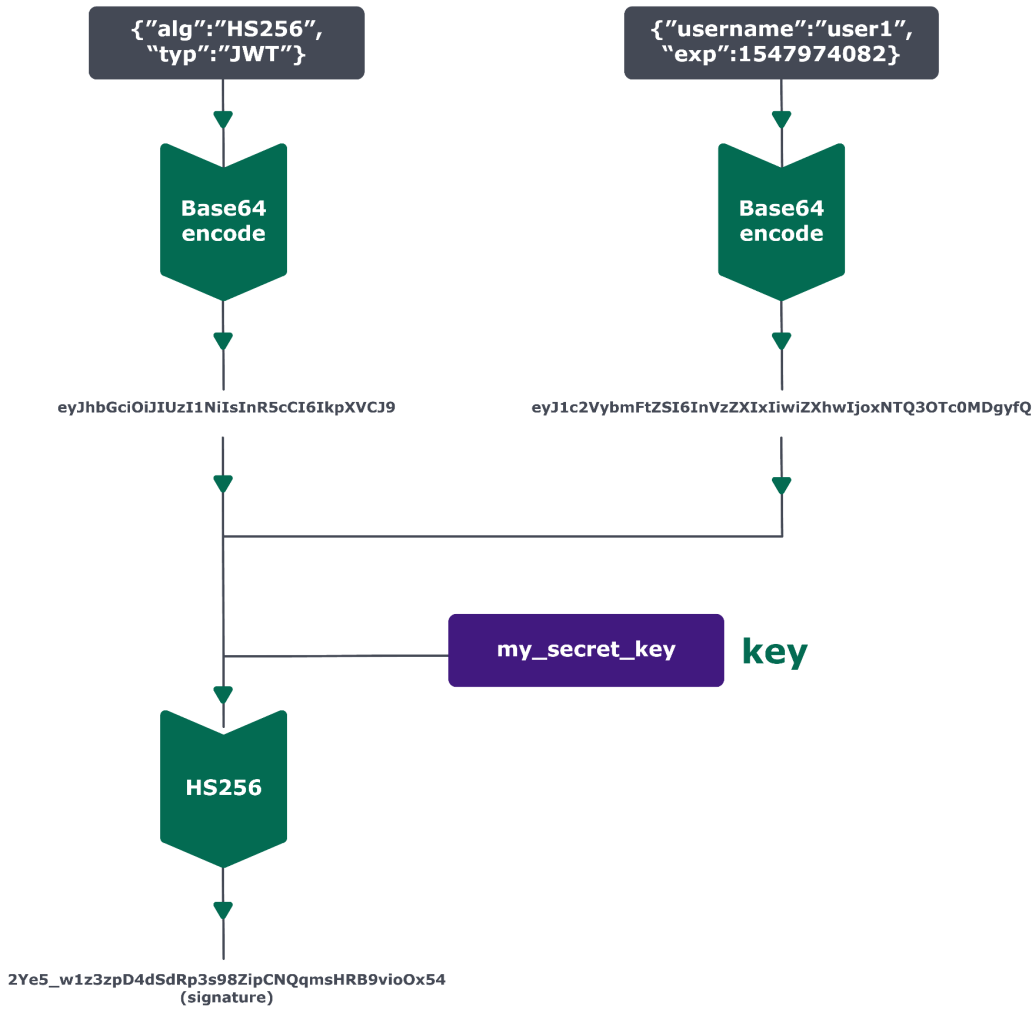
- Header: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9`
- Payload: `eyJ1c2VybmFtZSI6InVzZXIxIiwiZXhwIjoxNTQ3OTc0MDgyfQ`
- Signature: `2Ye5_w1z3zpD4dSGdRp3s98ZipCNQqmsHRB9vioOx54`

Um exemplo de envio de um JWT via header em uma requisição HTTP:

Copiar

```
GET /foo/bar HTTP/1.1
 Host: www.exemplo.com
 Authorization: Bearer (Header em base64).(Payload em
base64).(Signature em base64)
```

Ou seja, por ser algo que é transmitido via HTTP, JWT pode ser usado com qualquer linguagem que suporte os requisitos mínimos para gerar o token e enviar uma requisição HTTP, como, por exemplo, Java, C#, PHP ou Python.
O diagrama abaixo ilustra como um JWT é gerado a partir do header, do payload e da chave secreta.



Geração de um JWT

Show me the code!

Chegou a hora tão esperada. Vamos colocar a mão na massa!
Nesse exemplo, vamos trabalhar com as seguintes tecnologias:

- Nodejs;
- Express;
- Postman;
- MongoDB;
- JWT.

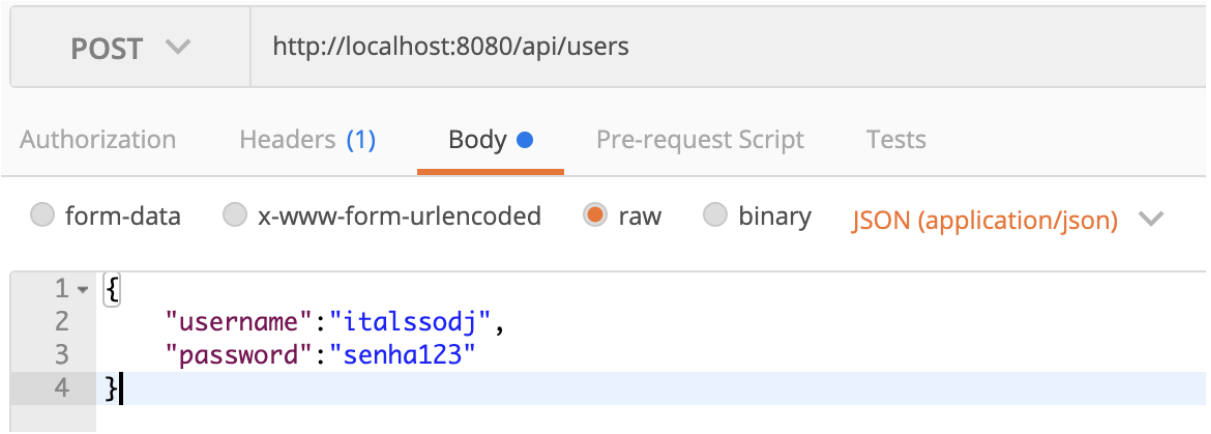
Para começar, vamos usar um projeto base. Esse projeto é, basicamente, uma API Express sem autenticação JWT. Neste link , você pode encontrar o código e as instruções para baixá-lo e executá-lo, além de uma explicação de como o projeto base funciona.
Nota : Após clonar o projeto, não se esqueça de colocar o endereço do MongoDB no arquivo `models/connection.js` , na linha 3. O endereço da sua instância local do MongoDB ficará disponível assim que você executar o `mongo` no seu terminal. Normalmente, esse endereço é `mongodb://127.0.0.1:27017` .

Testando nossa API

Imagine que esse é um serviço real que você usará em produção. Tenha isso mente, pois, nesses testes, vamos pegar alguns problemas que o JWT nos ajudará a resolver!
Para testar nossa aplicação, vamos usar o Postman , um serviço fácil e simples para fazer requisições HTTP.
Caso você ainda não tenha usado o Postman, assista a este vídeo que explica o funcionamento dessa ferramenta:

VIDEO: 28.1 - Postaman 11:51

Aprendido o uso do Postman, vamos ao trabalho!
Para começar a usar a nossa plataforma, precisamos criar um usuário. Para isso, faremos uma requisição POST para o endpoint `/api/users` , passando um nome de usuário e senha:

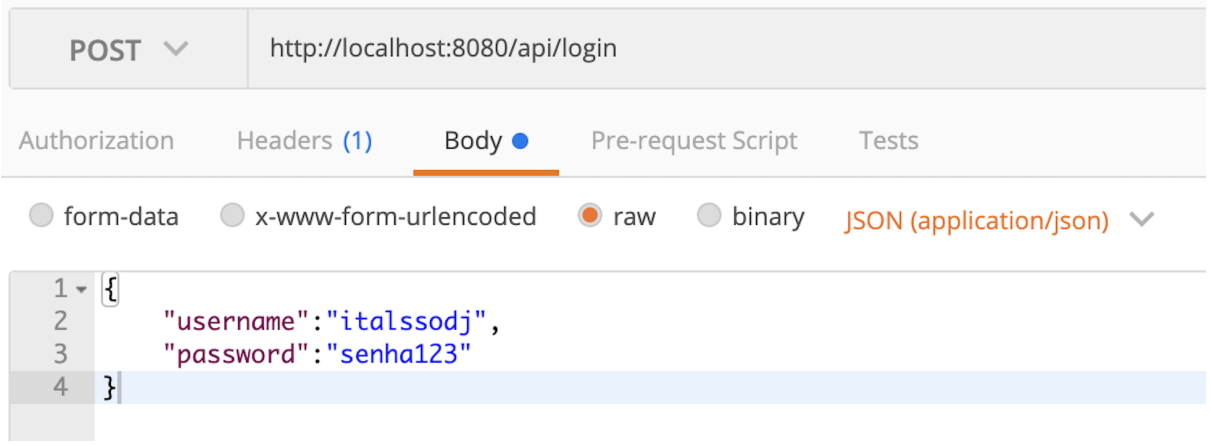


Após enviar essa requisição, obtemos a seguinte resposta:

Copiar

```
{
  "message": "Novo usuário criado com sucesso",
  "user": "italssodj"
}
```

Até aí, tudo certo. Não precisamos de autenticação para criar um usuário, mas, para consultar as nossas postagens no blog, precisamos sim! Então, vamos fazer o login. Para isso, fazemos uma requisição `POST` para o endpoint `/api/login` , passando o nome de usuário e senha usados no cadastro:

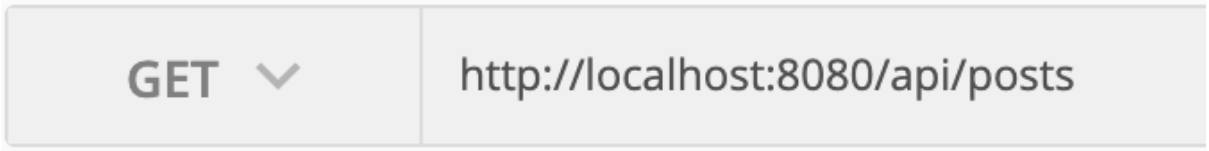


Após enviar essa requisição, recebemos a seguinte resposta:

Copiar

```
{
  "message": "Login efetuado com sucesso"
}
```

Legal, estamos logados. Agora já podemos pegar as postagens do nosso blog! Fazemos uma requisição `GET` para o endpoint `/api/posts/` :



E recebemos os posts como resposta:

Copiar

```
{
    "mockPosts": [
        {
            "title": "título fake",
            "content": "conteúdo conteúdo conteúdo conteúdo
conteúdo"
        },
        {
            "title": "título fake",
            "content": "conteúdo conteúdo conteúdo conteúdo
conteúdo"
        },
        {
            "title": "título fake",
            "content": "conteúdo conteúdo conteúdo conteúdo
conteúdo"
        },
        {
            "title": "título fake",
            "content": "conteúdo conteúdo conteúdo conteúdo
conteúdo"
        }
    ]
}
```

Agora, note uma coisa: quando formos utilizar a API com nosso front-end, como é que sabemos que estamos autenticados? Podemos fazer essa verificação no front-end e, caso a requisição de login retorne "sucesso", fazemos uma requisição para obter os posts. Mas e se alguém inspecionar as requisições do browser e descobrir o endpoint `/api/post` ? Essa pessoa vai poder acessar dados que não deveria. Além disso, se o browser for fechado, terá que logar novamente toda vez que precisar usar a API? E se for um sistema de banco, em que só se pode ficar online por um determinado tempo, como saberemos que a sessão expirou? No back-end, ao chegar uma requisição para `/api/posts` , como fazemos para retornar somente os posts de quem requisitou? Se o acesso for a um

recurso que requer um nível de autorização mais elevado, como fazemos para saber se a pessoa é, por exemplo, um admin?
São MUITAS dúvidas, mas calma. É aí que o JWT entra. Agora vamos alterar um pouco nossa API para adicionar autenticação via JWT. No final, vamos poder saber se a pessoa usuária está de fato autenticada, quem essa pessoa é e definir um tempo de sessão para ela. Caso essa pessoa esteja autenticada e um JWT válido seja apresentado no header Authorization, ela será autorizada a acessar diversas rotas dentro da aplicação, de acordo com suas credenciais, sem a necessidade de uma nova autenticação a cada requisição.

Implementando JWT

Para começar, vamos instalar o pacote `jsonwebtoken` . Ele é quem será responsável por gerar e validar os tokens para nós:

Copiar
```
npm install jsonwebtoken
```

Agora, vamos editar o arquivo `controllers/login.js` . Lá, vamos trabalhar na geração do nosso JWT e adicionar os seguintes trechos de código:

Copiar
```
// const User = require('../models/user');
const jwt = require('jsonwebtoken');

/* Sua chave secreta. É com ela que os dados do seu usuário serão encriptados.
   Em projetos reais, armazene-a numa variável de ambiente e tenha cuidado com ela, pois só quem tem acesso
   a ela poderá criar ou alterar tokens JWT. */
const secret = 'seusecretdetoken';

// module.exports = async (req, res) => {
//   try {
//     const username = req.body.username;
//     const password = req.body.password;

//     if (!username || !password)
//       return res
//         .status(401)
//         .json({ message: 'É necessário usuário e senha para fazer login' });
```

```
//     const user = await User.findUser(username);

//     if (!user || user.password !== password)
//       return res
//         .status(401)
//         .json({ message: 'Usuário não existe ou senha inválida'
});

    /* Criamos uma config básica para o nosso JWT, onde:
      expiresIn -> significa o tempo pelo qual esse token será
válido;
      algorithm -> algoritmo que você usará para assinar sua
mensagem
                   (lembra que falamos do HMAC-SHA256 lá no começo?).
*/

    /* A propriedade expiresIn aceita o tempo de forma bem
descritiva. Por exemplo: '7d' = 7 dias. '8h' = 8 horas. */
    const jwtConfig = {
      expiresIn: '7d',
      algorithm: 'HS256',
    };

    /*
      Aqui é quando assinamos de fato nossa mensagem com a nossa
"chave secreta".
      Mensagem essa que contém dados do seu usuário e/ou demais
dados que você
      quiser colocar dentro de "data".
      O resultado dessa função será equivalente a algo como:
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7Il9pZCI6IjVlNTQ1OTB
iYTQ5NDQ4ZjdlNWZhNzNjMCIsInVzZXJuYW1lIjoiaXRhbHNzb2RqIiwicGFzc3dvcmQ
iOiJzZW5oYTEyMyIsIl9fdiI6MH0sImlhdCI6MTU4MjU4NzMyNywiZXhwIjoxNTg0Nzc
0NzE0OTA4fQ.UdSZi7K105aaVnoKSW-dnw-Kv7H3oKMtE9xv4jwyfSM
      */
    const token = jwt.sign({ data: user }, secret, jwtConfig);

    /* Por fim, nós devolvemos essa informação ao usuário. */
    res.status(200).json({ token });
//   } catch (e) {
//     return res.status(500).json({ message: 'Erro interno', error:
e });
//   }
```

```
// };
```

Feito isso, nós já podemos nos autenticar de verdade, não é mesmo? Ao fazer uma nova requisição `POST` para `/api/login` , passando nome de usuário e senha corretos, obtemos um resultado semelhante ao seguinte:

Copiar

```
{
  "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjp7Il9pZCI6IjVlYTIyNT
U3ODNkNGJjNjExNzNjZDFlNiIsInVzZXJuYW1lIjoidHVsaW9zdGFybGluZyIsInBhc3
N3b3JkIjoiMTIzNDU2IiwiX192IjowfSwiaWF0IjoxNTg3Njg1MTAwLCJleHAiOjE1OD
k4Nzc1ODU2MTR9.Y0JzzSuSwrl2OXugqDd0UxY7D0g0HpW3gglSLdop4KU"
}
```

Eis o nosso token! É ele que vamos ficar transitando pra lá e pra cá, então ele precisa ser guardado! Mas caso ele seja perdido, não se preocupe; é só gerar outro token. 🙂
Agora temos que usar esse token de alguma forma, não é mesmo? Para isso, vamos criar uma pasta chamada `auth` dentro do diretório `api` ; e, dentro dela, um arquivo chamado `validateJWT.js` .
Esse arquivo conterá uma função que será usada como middleware para as nossas requisições, validando todas as rotas em que nós solicitarmos autenticação.

Copiar

```
// validateJWT.js
const jwt = require('jsonwebtoken');
const model = require('../../models/user');

/* Mesma chave privada que usamos para criptografar o token.
   Agora, vamos usá-la para descriptografá-lo.
   Numa aplicação real, essa chave jamais ficaria hardcoded no
código assim,
   e muitos menos de forma duplicada, mas aqui só estamos
interessados em
   ilustrar seu uso ;) */
const segredo = 'seusecretdetoken';

module.exports = async (req, res, next) => {
  /* Aquele token gerado anteriormente virá na requisição através do
     header Authorization em todas as rotas que queremos que
     sejam autenticadas. */
  const token = req.headers['authorization'];
```

```
  /* Caso o token não seja informado, simplesmente retornamos
     o código de status 401 - não autorizado. */
  if (!token) {
    return res.status(401).json({ error: 'Token não encontrado' });
  }

  try {
    /* Através o método verify, podemos validar e decodificar o
nosso JWT. */
    const decoded = jwt.verify(token, segredo);
    /*
      A variável decoded será um objeto equivalente ao seguinte:
      {
        data: {
          _id: '5e54590ba49448f7e5fa73c0',
          username: 'italssodj',
          password: 'senha123'
        },
        iat: 1582587327,
        exp: 1584774714908
      }
    */

    /* Caso o token esteja expirado, a própria biblioteca irá
retornar um erro,
       por isso não é necessário fazer validação do tempo.
       Caso esteja tudo certo, nós então buscamos o usuário na base
para obter seus dados atualizados */
    const user = await model.findUser(decoded.data.username);

    /* Não existe um usuário na nossa base com o id informado no
token. */
    if (!user) {
      return res
        .status(401)
        .json({ message: 'Erro ao procurar usuário do token.' });
    }

    /* O usuário existe! Colocamos ele em um campo no objeto req.
       Dessa forma, o usuário estará disponível para outros
middlewares que
       executem em sequência */
    req.user = user;
```

```
    /* Por fim, chamamos o próximo middleware que, no nosso caso,
       é a própria callback da rota. */
    next();
  } catch (err) {
    return res.status(401).json({ message: err.message });
  }
};
```

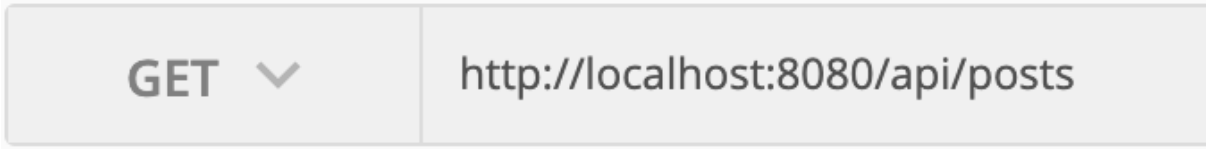
No arquivo `api/server.js` , onde definimos as rotas, usamos esse middleware para adicionar autenticação na nossa rota de listagem de posts.

Copiar

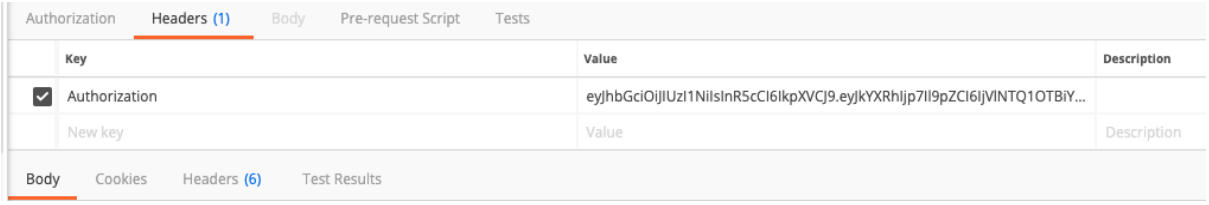
```
// const express = require('express');
// const bodyParser = require('body-parser');
// const routes = require('./routes');

/* Aqui, importamos nossa função que valida se o usuário está ou não
autenticado */
   const validateJWT = require('./auth/validateJWT');

// const port = process.env.PORT || 8080;

// const app = express();

// app.use(bodyParser.urlencoded({ extended: false }));
// app.use(bodyParser.json());

// const apiRoutes = express.Router();

/* E a usamos como middleware na nossa rota, colocando-a antes do
nosso controller. */
   apiRoutes.get('/api/posts', validateJWT, routes.getPosts);
// apiRoutes.post('/api/users', routes.createUsers);
// apiRoutes.post('/api/login', routes.login);

// app.use(apiRoutes);

// app.listen(port);
// console.log('conectado na porta ' + port);
```

Note que não queremos autenticar o login e nem criação de usuários, pois precisamos deles para o processo de autenticação! Se houvesse outras

rotas protegidas na nossa aplicação, usaríamos o middleware nelas também!
Agora que já estamos logados, vamos requisitar nossos posts!



Você deve estar pensando: ué! Já fizemos o login, então, por que não podemos pegar os posts? Olhe com mais atenção para a resposta da API. Ela está dizendo que o token não foi informado. Nós conseguimos um token através do endpoint de login, mas não fizemos nada com ele. Nesse caso, vamos mandar o token para a API via `Headers` , que são informações extras que podemos passar em uma requisição.



Adicionamos um header chamado `Authorization` porque é o que nosso middleware espera. Se não se lembra, dê uma olhada de novo no arquivo `/api/auth/validateJWT.js` .
Feito isso, é só mandar bala na requisição e ser feliz!

Copiar

```
{
  "mockPosts": [
    {
      "title": "titulo fake",
      "content": "conteudo conteudo conteudo conteudo conteudo "
    },
    {
      "title": "titulo fake",
      "content": "conteudo conteudo conteudo conteudo conteudo "
```

```
    },
    {
      "title": "titulo fake",
      "content": "conteudo conteudo conteudo conteudo conteudo "
    },
    {
      "title": "titulo fake",
      "content": "conteudo conteudo conteudo conteudo conteudo "
    }
  ]
}
```

Voltamos a conseguir recuperar nossos posts. Mas, antes de terminarmos, um último comentário sobre nossa API. Você notou que nossos posts são *fake* e são sempre os mesmos, independente do usuário logado, certo? Numa API real, buscaríamos esses posts de um banco de dados, por exemplo. Mas como faríamos para recuperar apenas os posts do usuário logado?
Lembra-se de que o middleware de autenticação recupera o usuário do banco de dados e o coloca no `req` ? Esse objeto é o mesmo que é passado para todos os middlewares e para a callback da rota. Como o middleware de autenticação é executado antes das funções dos controllers, `req` conterá o usuário logado quando o controller em `/controllers/posts` for executado, e poderíamos utilizá-lo para fazer uma consulta ao banco de dados que trouxesse somente seus posts. Para confirmar isso, basta colocar um `console.log` dentro do controller:

Copiar

```
module.exports = (req, res, next) => {
  console.log(req.user);
  res.status(200).json({ mockPosts });
};
```

Você deverá ver algo assim, no terminal onde executou a API:

Copiar

```
{
  _id: 5ea2255783d4bc61173cd1e6,
  username: 'italssodj',
  password: 'senha123'
}
```

Exercícios

Hora de pôr a mão na massa!
back-end

Antes de começar: versionando seu código

Para versionar seu código, utilize o seu repositório de exercícios. 😉
Abaixo você vai ver exemplos de como organizar os exercícios do dia em uma `branch`, com arquivos e `commits` específicos para cada exercício. Você deve seguir este padrão para realizar os exercícios a seguir.

1. Abra a pasta de exercícios:
2. Copiar
3. `$ cd ~/trybe-exercicios`
4. Certifique-se de que está na branch main e ela está sincronizada com a remota. Caso você tenha arquivos modificados e não comitados, deverá fazer um commit ou checkout dos arquivos antes deste passo.
5. Copiar

```
$ git checkout main
```

6. `$ git pull`
7. A partir da main, crie uma `branch` com o nome `exercicios/27.1` *(bloco 27, dia 1)*
8. Copiar
9. `$ git checkout -b exercicios/27.1`
10. Caso seja o primeiro dia deste módulo, crie um diretório para ele e o acesse na sequência:
11. Copiar

```
$ mkdir back-end
```

12. `$ cd back-end`
13. Caso seja o primeiro dia do bloco, crie um diretório para ele e o acesse na sequência:
14. Copiar

```
$ mkdir bloco-27-autenticacao-e-upload-de-arquivos
```

15. `$ cd bloco-27-autenticacao-e-upload-de-arquivos`
16. Crie um diretório para o dia e o acesse na sequência:
17. Copiar

```
$ mkdir dia-1-nodejs-jwt-json-web-token
```

18. `$ cd dia-1-nodejs-jwt-json-web-token`
19. Os arquivos referentes aos exercícios deste dia deverão ficar dentro do diretório `~/trybe-exercicios/back-end/block-27-autenticacao-e-upload-de-arquivos/dia-1-nodejs-jwt-json-web-token`. Lembre-se de fazer commits pequenos e com mensagens bem descritivas, preferencialmente a cada exercício resolvido.

    Verifique os arquivos alterados/adicionados:
20. Copiar

```
$ git status
On branch exercicios/27.1
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
```

21. `  modified:   exercicio-1`

    Adicione os arquivos que farão parte daquele commit:
22. Copiar

```
# Se quiser adicionar os arquivos individualmente
$ git add caminhoParaArquivo

# Se quiser adicionar todos os arquivos de uma vez, porém, atente-se
```

```
para não adicionar arquivos indesejados acidentalmente
```

23. `$ git add --all`

    Faça o commit com uma mensagem descritiva das alterações:
24. Copiar
25. `$ git commit -m "Mensagem descrevendo alterações"`
26. Você pode visualizar o log de todos os commits já feitos naquela branch com `git log`.
27. Copiar

```
$ git log
commit 100c5ca0d64e2b8649f48edf3be13588a77b8fa4 (HEAD ->
exercicios/27.1)
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 17:48:01 2019 -0300

    Exercicio 2 - mudando o evento de click para mouseover, tirei o
alert e coloquei pra quando clicar aparecer uma imagem do lado
direito da tela

commit c0701d91274c2ac8a29b9a7fbe4302accacf3c78
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 16:47:21 2019 -0300

    Exercicio 2 - adicionando um alert, usando função e o evento
click

commit 6835287c44e9ac9cdd459003a7a6b1b1a7700157
Author: Tryber Bot <tryberbot@betrybe.com>
Date:   Fry Sep 27 15:46:32 2019 -0300
```

28. `    Resolvendo o exercício 1 usando eventListener`
29. Agora que temos as alterações salvas no repositório local precisamos enviá-las para o repositório remoto. No primeiro envio, a branch `exercicios/27.1` não vai existir no repositório remoto, então precisamos configurar o `remote` utilizando a opção `--set-upstream` (ou `-u`, que é a forma abreviada).
30. Copiar
31. `$ git push -u origin exercicios/27.1`

32. Após realizar o passo 9, podemos abrir a `Pull Request` a partir do link que aparecerá na mensagem do push no terminal, ou na página do seu repositório de exercícios no GitHub através de um botão que aparecerá na interface. Escolha a forma que preferir e abra a Pull Request. De agora em diante, você repetirá o fluxo a partir do passo 7 para cada exercício adicionado, porém como já definimos a branch remota com `-u` anteriormente, agora podemos simplificar os comandos para:
33. Copiar

```
# Quando quiser enviar para o repositório remoto
$ git push

# Caso você queria sincronizar com o remoto, poderá utilizar apenas
$ git pull
```

34.
35. Quando terminar os exercícios, seus códigos devem estar todos commitados na branch `exercicios/27.1`, e disponíveis no repositório remoto do `GitHub`. Pra finalizar, compartilhe o link da Pull Request no canal de Code Review para a monitoria e/ou colegas revisarem. Faça review você também, lembre-se que é muito importante para o seu desenvolvimento ler o código de outras pessoas. 👊🤜

Agora, a prática

Antes de começar, crie um novo projeto chamado `hello-jwt` utilizando o comando `npm init @tryber/backend hello-jwt` , aceitando as opções padrão.

1. Crie um endpoint `POST /login`
2. O endpoint deve receber os seguintes dados no corpo da requisição:

Copiar

```
{
  "username": "someUsername",
  "password": "somePassword"
}
```

2. Caso `username` e `password` sejam válidos, retorne um token que atenda às seguintes especificações:

    - Expira em uma hora;
    - Contém, no payload, o nome de usuário informado na request;
    - Contém, no payload, uma propriedade `admin` , com o valor `false` .
3. Para retornar o token, utilize o seguinte formato no corpo da resposta:

Copiar
```
{
  "token": "<JWT aqui>"
}
```

4. Para que `username` seja válido, seu valor precisa ser uma string alfanumérica de, pelo menos, 5 caracteres.
5. Para que `password` seja válido, seu valor precisa ser uma string de, pelo menos, 5 caracteres.
6. Altere o endpoint `POST /login` :
7. Caso `username` seja `admin` e `password` seja `s3nh4S3gur4???` , a chave `admin` no payload do token gerado deve ter o valor `true`
8. Crie o endpoint `/GET /users/me`
9. O endpoint só pode ser acessado por pessoas autenticadas
10. Para realizar a autenticação, a requisição deve conter o header `Authorization` , cujo valor deve ser um token válido
11. Caso o token não exista, retorne o status `401 Unauthorized` , com o seguinte corpo da resposta:

Copiar
```
{
  "error": {
    "message": "Token not found"
  }
}
```

4. Caso aconteça um erro ao validar o token, retorne o status `401 Unauthorized` com o seguinte conteúdo no corpo:

Copiar
```
{
  "error": {
    "message": "<mensagem de erro da biblioteca>"
  }
}
```

5. Caso o token seja válido, retorne o status `200 OK` e, no corpo da resposta, o nome de usuário ao qual aquele token pertence e o valor da propriedade `admin` , no seguinte formato:

Copiar
```
{
  "username": "nome de usuario do token",
  "admin": true || false
}
```

6. Utilize um middleware exclusivo para a autenticação. Armazene-o no arquivo `middlewares/auth.js`
7. Crie o endpoint `/GET /top-secret`
8. O endpoint só pode ser acessado por pessoas autenticadas.
9. Apenas tokens contendo, no payload, a propriedade `admin` com o valor `true` têm autorização para acessar esse endpoint.
10. Caso o token não exista, retorne o status `401 Unauthorized` , com o seguinte corpo da resposta:

Copiar
```
{
  "error": {
    "message": "Token not found"
  }
}
```

4. Caso aconteça um erro ao validar o token, retorne o status `401 Unauthorized` com o seguinte conteúdo no corpo:

Copiar
```
{
  "error": {
    "message": "<mensagem de erro da biblioteca>"
  }
}
```

5. Caso o token seja válido, mas a propriedade `admin` do payload não seja `true` , retorne o status `403 Forbidden` e o seguinte JSON:

Copiar
```
{
  "error": {
    "message": "Restricted access"
  }
```

```
}
```

6. Caso o token seja válido e o payload contenha `admin` com o valor `true` , retorne o seguinte JSON:

Copiar
```
{
  "secretInfo": "Peter Parker é o Homem-Arannha"
}
```

7. Para validar se a pessoa é admin, crie um novo middleware no arquivo `middlewares/admin.js` .

Bônus

1. Crie o endpoint `POST /signup`
2. O endpoint deve aceitar o seguinte JSON no corpo da requisição:

Copiar
```
{
  "username": "MariaCecília_Souza92",
  "password": "%9!%e"c0c5w,q%%h9n3k"
}
```

2. Para validar os campos, considere os mesmos critérios do endpoint `POST /login` ;
3. Caso `username` já exista, retorne o status `409 Conflict` e o seguinte JSON:

Copiar
```
{
   "error": { "message": "user already exists" }
}
```

4. Caso os campos sejam válidos, armazene os dados no arquivo `models/data/users.json` ;
5. Ao armazenar os dados recebidos, adicione a propriedade `admin` , que terá seu valor determinado da seguinte forma:
    - Obtenha um número aleatório de 1 a 100 com o seguinte trecho de código: `Math.floor(Math.random() * 100)` ;

- Caso o número aleatório seja maior que `50` , `admin` deve ser `true` ;
- Caso o número aleatório seja menor ou igual a `50` , `admin` deve ser `false` .

6. Após armazenar os novos dados, retorne um token que expire em uma hora e contenha `username` e `admin` no payload. Utilize o seguinte formato na resposta:

Copiar

```
{
    "token": "<token gerado aqui>"
}
```

2. Altere o endpoint de login
3. Antes de gerar o token, verifique se o nome de usuário e a senha informados existem no arquivo `users.json` ;
4. Não permita mais o login do usuário `admin` com a senha fixa.
5. Informe, na propriedade `admin` do payload do token, o mesmo valor da propriedade `admin` que está armazenado para aquela pessoa.

Gabarito dos exercícios

Lembre-se que os exercícios servem para consolidar o seu aprendizado e te ajudar a identificar as suas dúvidas! Portanto, é esperado que você trave em algum momento. Somente olhe o gabarito depois de tentar resolver o problema por algum tempo! Garanta que você entendeu tudo que está aqui, explique para "a parede" para ter certeza e, se houver dificuldade, tire sua dúvida conosco!

Exercícios

Exercício 1

Crie um endpoint `POST /login`

1. O endpoint deve receber os seguintes dados no corpo da requisição:

Copiar

```
{
```

```
  "username": "someUsername",
  "password": "somePassword"
}
```

2. Caso `username` e `password` sejam válidos, retorne um token que atenda às seguintes especificações:
    - Expira em uma hora;
    - Contém, no payload, o nome de usuário informado na request;
    - Contém, no payload, uma propriedade `admin` , com o valor `false` .
3. Para retornar o token, utilize o seguinte formato no corpo da resposta:

Copiar
```
{
  "token": "<JWT aqui>"
}
```

4. Para que `username` seja válido, seu valor precisa ser uma string alfanumérica de, pelo menos, 5 caracteres.
5. Para que `password` seja válido, seu valor precisa ser uma string de, pelo menos, 5 caracteres.

Resolução

1. Primeiro, precisamos criar um novo controller na pasta `controllers` . Crie o arquivo `controllers/login.js` :

Copiar
```
// controllers/login.js
module.exports = async (req, res, next) => {
  //
}
```

2. O próximo passo é utilizamos o `Joi` para validar a requisição:

Copiar
```
// controllers/login.js
const Joi = require('joi');

const validateBody = (body) =>
  /* Utilizamos o Joi para validar o schema do body */
```

```
  Joi.object({
    username: Joi.string().min(5).alphanum().required(),
    password: Joi.string().min(5).required(),
  }).validate(body);

module.exports = async (req, res, next) => {
  /* Construímos um schema do Joi */
  const { error } = validateBody(req.body);

  /* Caso ocorra erro na validação do Joi, passamos esse */
  /* erro para o express, que chamará nosso middleware de erro */
  if (error) return next(error);
};
```

3. Antes de prosseguir, precisamos instalar a biblioteca de JWT que vamos utilizar:

Copiar
```
npm i jsonwebtoken
```

4. Agora, precisamos criar o token JWT:

Copiar
```
// controllers/login.js
// const Joi = require('joi');
const jwt = require('jsonwebtoken');

const { JWT_SECRET } = process.env;

//const validateBody = (body) =>
//  /* Utilizamos o Joi para validar o schema do body */
//  Joi.object({
//    username: Joi.string().min(5).alphanum().required(),
//    password: Joi.string().min(5).required(),
//  }).validate(body);

// module.exports = async (req, res, next) => {
//   const { error } = validateBody(req.body);
//   /* Caso ocorra erro na validação do Joi, passamos esse */
//   /* erro para o express, que chamará nosso middleware de erro */
//   if (error) return next(error);

  const payload = {
    username: req.body.username,
```

```
    admin: false,
  };

  const token = jwt.sign(payload, JWT_SECRET, {
    expiresIn: '1h',
  });

  res.status(200).json({ token });
// };
```

5. Adicione a variável `JWT_SECRET` ao arquivo `.env`

Copiar

```
# PORT=3000
JWT_SECRET=meuSegredoSuperSegreto
```

6. Adicione o controller de login ao arquivo `controllers/index.js` :

Copiar

```
// controllers/index.js
// const ping = require('./ping');
const login = require('./login');

// module.exports = {
//   ping,
  login,
// };
```

7. Por último, registramos o endpoint no express:

Copiar

```
// index.js

// require('dotenv').config();
// const express = require('express');
// const cors = require('cors');
// const bodyParser = require('body-parser');

// const { PORT } = process.env;

// const controllers = require('./controllers');
// const middlewares = require('./middlewares');

// const app = express();
```

```
// app.use(
//   cors({
//     origin: `http://localhost:${PORT}`,
//     methods: ['GET', 'POST', 'PUT', 'DELETE'],
//     allowedHeaders: ['Authorization'],
//   })
// );

// app.use(bodyParser.json());
// app.use(bodyParser.urlencoded({ extended: true }));

// app.get('/ping', controllers.ping);
app.post('/login', controllers.login);
// app.use(middlewares.error);

// app.listen(PORT, () => {
//   console.log(`App listening on port ${PORT}`);
// });
```

Exercício 2

Altere o endpoint `POST /login` :

1. Caso `username` seja `admin` e `password` seja `s3nh4S3gur4???` , a chave `admin` no payload do token gerado deve ter o valor `true`

Resolução

1. Precisamos adicionar uma condição especial para esse caso. Para isso, altere o arquivo `controllers/login.js` :

Copiar

```
const Joi = require('joi');
// const jwt = require('jsonwebtoken');

// const { JWT_SECRET } = process.env;
//const validateBody = (body) =>
//  /* Utilizamos o Joi para validar o schema do body */
//  Joi.object({
//    username: Joi.string().min(5).alphanum().required(),
//    password: Joi.string().min(5).required(),
```

```
//   }).validate(body);

// module.exports = async (req, res, next) => {
//   const { error } = validateBody(req.body);
//   /* Caso ocorra erro na validação do Joi, passamos esse */
//   /* erro para o express, que chamará nosso middleware de erro */
//   if (error) return next(error);

  /* Se o login for admin e a senha estiver incorreta */
  if (req.body.username === 'admin' && req.body.password !==
's3nh4S3gur4???') {
    /* Criamos um novo objeto de erro */
    const err = new Error('Invalid username or password');
    /* Adicionamos o status `401 Unauthorized` ao erro */
    err.statusCode = 401;
    /* Passmos o erro para o express, para que seja tratado pelo
middleware de erro */
    return next(err);
  }

  /* Definimos admin como true se username e password estiverem
corretos */
  const admin = req.body.username === 'admin' && req.body.password
=== 's3nh4S3gur4???';

//   const payload = {
//     username: req.body.username,
    /* Passamos a utilizar o valor da variável `admin` */
    /* para determinar o valor do campo `admin` no payload do token
*/
    admin,
//   };

//   const token = jwt.sign(payload, JWT_SECRET, {
//     expiresIn: '1h',
//   });

//   res.status(200).json({ token });
// };
```

Exercício 3

Crie o endpoint `/GET /users/me`

1. O endpoint só pode ser acessado por pessoas autenticadas
2. Para realizar a autenticação, a requisição deve conter o header `Authorization` , cujo valor deve ser um token válido
3. Caso o token não exista, retorne o status `401 Unauthorized` , com o seguinte corpo da resposta:

Copiar

```
{
  "error": {
    "message": "Token not found"
  }
}
```

4. Caso aconteça um erro ao validar o token, retorne o status `401 Unauthorized` com o seguinte conteúdo no corpo:

Copiar

```
{
  "error": {
    "message": "<mensagem de erro da biblioteca>"
  }
}
```

5. Caso o token seja válido, retorne o status `200 OK` e, no corpo da resposta, o nome de usuário ao qual aquele token pertence e o valor da propriedade `admin` , no seguinte formato:

Copiar

```
{
  "username": "nome de usuario do token",
  "admin": true || false
}
```

6. Utilize um middleware exclusivo para a autenticação. Armazene-o no arquivo `middlewares/auth.js`

Resolução

1. Vamos começar pela criação do middleware de autenticação. Crie o arquivo `middlewares/auth.js`

Copiar

```
// middlewares/auth.js
module.exports = (req, res, next) => {
  //
}
```

2. Agora, importamos o JWT e verificamos se o token foi enviado

Copiar

```
// middlewares/auth.js
const jwt = require('jsonwebtoken');

const { JWT_SECRET } = process.env;

// module.exports = (req, res, next) => {
  /* Buscamos o token no header `Authorization` */
  const token = req.headers.authorization;

  /* Caso o token não exista */
  if (!token) {
    /* Criamos um novo objeto de erro */
    const err = new Error('Token not found');
    /* Damos o status 401 ao erro */
    err.statusCode = 401;
    /* Enviamos o erro para ser tratado pelo middleware de erro */
    return next(err);
  }
// }
```

3. Caso o token exista, precisamos verificar se ele é válido. Altere o arquivo `middlewares/auth.js`

Copiar

```
// middlewares/auth.js
// const jwt = require('jsonwebtoken');

// const { JWT_SECRET } = process.env;

// module.exports = (req, res, next) => {
//   /* Buscamos o token no header `Authorization` */
//   const token = req.headers.authorization;

//   /* Caso o token não exista */
//   if (!token) {
//     /* Criamos um novo objeto de erro */
```

```
//     const err = new Error('Token not found');
//     /* Damos o status 401 ao erro */
//     err.statusCode = 401;
//     /* Enviamos o erro para ser tratado pelo middleware de erro
*/
//     return next(err);
//   }

  /* Realizamos uma tentativa de validar o token */
  try {
    /* Pedimos para que a bilioteca de JWT valide o token */
    const payload = jwt.verify(token, JWT_SECRET);

    /* Caso não ocorra nenhum erro, significa que o token é válido e
podemos continuar */

    /* Armazenamos os dados da pessoa no objeto de request */
    req.user = payload

    return next()
  } catch (err) {
    /* Caso haja algum erro ao validar o token, adicionamos o status
401 a esse erro */
    err.statusCode = 401;
    /* E enviamos o erro para ser processador pelo middleware de
erro. */
    return next(err);
  }
// };
```

4. Adicione o middleware de autenticação ao arquivo `middlewares/index.js` :

Copiar

```
// middlewares/index.js

const auth = require('./auth');
// const error = require('./error');

// module.exports = {
  auth,
//   error,
// };
```

5. Middleware pronto, podemos seguir para o controller. Crie o arquivo `controllers/me.js` :

Copiar

```
// controllers/me.js

module.exports = (req, res) => {
  /* Se chegamos até aqui, quer dizer que o middleware de
autenticação */
  /* foi executado, e adicionou as informações do token no objeto
`req`. */
  /* Podemos, então, extrair as propriedades que queremos de
`req.user` */
  const { username, admin } = req.user;

  /* Por fim, retornamos as informações */
  res.status(200).json({ username, admin });
};
```

6. Adicione o novo controller ao arquivo `controllers/index.js` :

Copiar

```
// controllers/index.js

const me = require('./me');
// const ping = require('./ping');
// const login = require('./login');

// module.exports = {
  me,
//   ping,
//   login,
// };
```

7. Por último, registramos o endpoint no arquivo `index.js` :

Copiar

```
// index.js

// app.use(bodyParser.urlencoded({ extended: true }));

// app.get('/ping', controllers.ping);
// app.post('/login', controllers.login);
app.get('/users/me', middlewares.auth, controllers.me);
```

```
// app.use(middlewares.error);
```

Exercício 4

Crie o endpoint `/GET /top-secret`

1. O endpoint só pode ser acessado por pessoas autenticadas.
2. Apenas tokens contendo, no payload, a propriedade `admin` com o valor `true` têm autorização para acessar esse endpoint.
3. Caso o token não exista, retorne o status `401 Unauthorized` , com o seguinte corpo da resposta:

Copiar

```
{
  "error": {
    "message": "Token not found"
  }
}
```

4. Caso aconteça um erro ao validar o token, retorne o status `401 Unauthorized` com o seguinte conteúdo no corpo:

Copiar

```
{
  "error": {
    "message": "<mensagem de erro da biblioteca>"
  }
}
```

5. Caso o token seja válido, mas a propriedade `admin` do payload não seja `true` , retorne o status `403 Forbidden` e o seguinte JSON:

Copiar

```
{
  "error": {
    "message": "Restricted access"
  }
}
```

6. Caso o token seja válido e o payload contenha `admin` com o valor `true` , retorne o seguinte JSON:

Copiar

```
{
  "secretInfo": "Peter Parker é o Homem-Aranha"
}
```

7. Para validar se a pessoa é admin, crie um novo middleware no arquivo `middlewares/admin.js` .

Resolução

1. Começamos criando um middleware que verifica se o token informado é do admin. Crie o arquivo `middlewares/admin.js` :

Copiar
```
// middlewares/admin.js

module.exports = (req, res, next) => {
  //
}
```

2. O middleware de admin deve ser executado após o middleware de autenticação. Sendo assim, vamos procurar por `req.user` para obter o valor da propriedade `admin` para aquela pessoa. Modifique o arquivo `middlewares/admin.js` :

Copiar
```
// middlewares/admin.js

// module.exports = (req, res, next) => {
    const { user } = req;

    /* Caso `req.user` não exista */
    if (!user) {
      /* Criamos um objeto de erro */
      const err = new Error('This endpoint requires
authentication');
      /* Atribuímos o status `401 Unauthorized` ao erro */
      err.statusCode = 401;
      /* E enviamos o erro para o middleware de erro */
      return next(err);
    }

    /* Caso o usuário não seja admin */
    if (!user.admin) {
```

```
      /* Criamos um novo erro com status `403 Forbidden` */
      const err = new Error('Restricted access');
      err.statusCode = 403;
      /* Enviamos o erro para ser processado no middleware de erros
*/
      return next(err);
    }

    /* Se nenhuma das condições acima forem verdadeiras, */
    /* a pessoa é admin e podemos continuar com a request */
    return next();
// }
```

3. Adicione o middleware admin ao arquivo `middlewares/index.js` :

Copiar

```
// middleware/index.js

// const auth = require('./auth');
const admin = require('./admin');
// const error = require('./error');

// module.exports = {
//   auth,
    admin,
//   error,
// };
```

4. Agora, vamos ao controller. Crie o arquivo `controllers/topSecret.js` :

Copiar

```
module.exports = (req, res) =>
  res.status(200).json({ secretInfo: 'Peter Parker é o
Homem-Arannha' });
```

5. Adicione o novo controller ao arquivo `controllers/index.js` :

Copiar

```
// const me = require('./me');
// const ping = require('./ping');
// const login = require('./login');
const topSecret = require('./topSecret');

// module.exports = {
```

```
//   me,
//   ping,
//   login,
     topSecret,
// };
```

6. Por fim, registre o novo endpoint no arquivo `index.js` :

Copiar

```
// index.js

// app.use(bodyParser.urlencoded({ extended: true }));

// app.get('/ping', controllers.ping);
// app.post('/login', controllers.login);
// app.get('/users/me', middlewares.auth, controllers.me);
app.get(
  '/top-secret',
  /* Middleware que valida o JWT e cria `req.user` */
  middlewares.auth,
  /* Middleware que verifica se a pessoa autenticada é admin */
  middlewares.admin,
  /* Controller do endpoint */
  controllers.topSecret
);

// app.use(middlewares.error);
```

Bônus

Exercício 1

1. Crie o endpoint `POST /signup`
2. O endpoint deve aceitar o seguinte JSON no corpo da requisição:

Copiar

```
{
  "username": "MariaCecília_Souza92",
  "password": "%9!%ec0c5w,q%%h9n3k"
}
```

2. Para validar os campos, considere os mesmos critérios do endpoint `POST /login` ;
3. Caso `username` já exista, retorne o status `409 Conflict` e o seguinte JSON:

Copiar
```
{
   "error": { "message": "user already exists" }
}
```

4. Caso os campos sejam válidos, armazene os dados no arquivo `models/data/users.json` ;
5. Ao armazenar os dados recebidos, adicione a propriedade `admin` , que terá seu valor determinado da seguinte forma:
    - Obtenha um número aleatório de 1 a 100 com o seguinte trecho de código: `Math.floor(Math.random() * 100)` ;
    - Caso o número aleatório seja maior que `50` , `admin` deve ser `true` ;
    - Caso o número aleatório seja menor ou igual a `50` , `admin` deve ser `false` .
6. Após armazenar os novos dados, retorne um token que expire em uma hora e contenha `username` e `admin` no payload. Utilize o seguinte formato na resposta:

Copiar
```
 {
    "token": "<token gerado aqui>"
  }
```

Resolução

1. Vamos começar criando um model para tratar da comunicação com o sistema de arquivos. Crie o arquivo `models/User.js` :

Copiar
```
// models/User.js

const path = require('path');
const fs = require('fs').promises;

/* Utilizamos o módulo `path` para calcular o caminho até o arquivo
`users.json` */
const DATA_PATH = path.join(__dirname, 'data', 'users.json');
```

```
/* Para obter todos os itens, lemos o arquivo `users.json, */
/* realizamos o parsing e retornamos o resultado */
const getAll = async () => fs.readFile(DATA_PATH,
'utf-8').then(JSON.parse);

/* Para alterar o arquivo `users.json`, recevemos um array, */
/* Convertemos o array em JSON e escrevemos o resultado no disco */
const writeAll = async (content) =>
  fs.writeFile(DATA_PATH, JSON.stringify(content));

/* Para encontrar um item, lemos todos os itens e utilizamos o
método `find` do array */
const findOne = (username) =>
  getAll().then((users) => users.find((user) => user.username ===
username));

/* Para criar um novo registro */
const create = (username, password, admin) =>
  /* Buscamos todos os itens */
  getAll()
    .then((users) => {
      /* Adicionamos o item novo */
      users.push({ username, password, admin });
      return users;
    })
    /* Armazenamos o Array no disco */
    .then(writeAll);

module.exports = {
  getAll,
  findOne,
  create,
};
```

Crie também o arquivo `models/data/users.json` contendo um array vazio:

Copiar

```
// models/data/users.json

[]
```

2. Agora, criamos um service para implementar as regras de negócio. Crie o arquivo `services/User.js` :

Copiar

```
// services/User.js

const model = require('../models/User');

const create = async (username, password) => {
  /* A primeira coisa que precisamos fazer
  é verificar se o username informado já existe */
  const userExists = await model.findOne(username);

  /* Caso o username já exista */
  if (userExists) {
    /* Retornamos um objeto de erro */
    return {
      error: {
        message: 'Username already exists',
        code: 'usernameExists',
      },
    };
  }

  /* Caso o username não exista, "rolamos o dado" para descobrir se
essa pessoa será admin */
  const admin = Math.floor(Math.random() * 100) > 50;

  /* Depois, armazenamos os dados no arquivo */
  await model.create(username, password, admin);

  /* Por fim, retornamos os dados da pessoa para o controller */
  /* Por motivos de segurança, não incluiremos a senha */
  return {
    username,
    admin,
  };
};

module.exports = {
  create,
};
```

3. Note que o exercício pede que, ao final, seja retornado um token. Nós já fazemos isso em outro lugar: no controller de login! Vamos,

então, mover o código responsável por gerar tokens para o service! Altere o arquivo `services/User.js` :

Copiar

```
const jwt = require('jsonwebtoken');
// const model = require('../models/User');

const { JWT_SECRET } = process.env;

/* Recebemos o valor de `admin` que, por padrão, é `false` */
const login = async (username, password, admin = false) => {
  /* Não precisamos validar os campos, pois o controler já faz isso
pra nós */

  /* Se o login for admin e a senha estiver incorreta */
  if (username === 'admin' && password !== 's3nh4S3gur4???') {
    /* Retornamos um objeto de erro */
    return {
      error: {
        message: 'Invalid username or password',
        code: 'invalidCredentials',
      },
    };
  }

  /* Caso a função login seja chamada com o parâmetro admin pré
definido, utilizamos esse parâmetro.
     Caso contrário, verificamos o nome de usuário e senha */
  const isAdmin = admin || (username === 'admin' && password ===
's3nh4S3gur4???');

  const payload = {
    username,
    /* Passamos a utilizar o valor da variável `admin` */
    /* para determinar o valor do campo `admin` no payload do token
*/
    admin: isAdmin,
  };

  const token = jwt.sign(payload, JWT_SECRET, {
    expiresIn: '1h',
  });
```

```
  return { token };
};

// const create = async (username, password) => {
//...
//   return {
//     username,
//     admin,
//   };
// };

// module.exports = {
//   create,
    login,
// };
```

4. Agora, alteramos o controller de login para que utilize o service que acabamos de criar. Altere o arquivo `controllers/login.js` :

Copiar

```
// controllers/login.js

const Joi = require('joi');

const service = require('../services/User');

const validateBody = (body) =>
  /* Utilizamos o Joi para validar o schema do body */
  Joi.object({
    username: Joi.string().min(5).alphanum().required(),
    password: Joi.string().min(5).required(),
  }).validate(body);

module.exports = async (req, res, next) => {
  const { error } = validateBody(req.body);

  /* Caso ocorra erro na validação do Joi, passamos esse */
  /* erro para o express, que chamará nosso middleware de erro */
  if (error) return next(error);

  const { username, password } = req.body;

  /* Pedimos para o service gerar o token */
```

```
  const { error: serviceError, token } = await
service.login(username, password);

  /* Caso ocorra um erro do tipo `invalidCredentials`,
     retornamos um novo erro com status `401 unauthorized` */
  if (serviceError && serviceError.code === 'invalidCredentials') {
    return next({ statusCode: 401, message: serviceError.message });
  }

  /* Caso haja qualquer outro erro, acionamos o middleware de erro
para obter uma mensagem genérica */
  if (serviceError) {
    return next(serviceError);
  }

  /* Por fim, caso nenhum erro tenha ocorrido, retornamos o token */
  res.status(200).json({ token });
};
```

5. Agora precisamos alterar nosso método `create` no service para que ele utilize o método login também. Altere o arquivo `services/User.js` :

Copiar

```
// ...
// const create = async (username, password) => {
//   /* A primeira coisa que precisamos fazer
//   é verificar se o username informado já existe */
//   const userExists = await model.findOne(username);

//   /* Caso o username já exista */
//   if (userExists) {
//     /* Retornamos um objeto de erro */
//     return {
//       error: {
//         message: 'Username already exists',
//         code: 'usernameExists',
//       },
//     };
//   }

//   /* Caso o username não exista, "rolamos o dado" para descobrir
se essa pessoa será admin */
```

```
//   const admin = Math.floor(Math.random() * 100) > 50;

//   /* Depois, armazenamos os dados no arquivo */
//   await model.save({ username, password, admin });

    /* Por fim, retornamos o token  */
    return login(username, password, admin);
// };
// ...
```

6. Já que precisamos reutilizar a lógica de validação que usamos no controller de login, vamos exportar essa lógica para outro arquivo. Crie o arquivo `controllers/utils/validateCredentials.js` :

Copiar

```
// controllers/utils/validateCredentials.js

const Joi = require('joi');

const validateCredentials = (body) =>
  /* Utilizamos o Joi para validar o schema do body */
  Joi.object({
    username: Joi.string().min(5).alphanum().required(),
    password: Joi.string().min(5).required(),
  }).validate(body);

module.exports = {
  validateCredentials,
};
```

7. Agora, alteramos o login para utilizar a função `validateCredentials` . Modifique o arquivo `controllers/login.js` :

Copiar

```
// controllers/login.js

/* Removemos o import do Joi */
// const service = require('../services/User');
const { validateCredentials } =
require('./utils/validateCredentials');

// module.exports = async (req, res, next) => {
  const { error } = validateCredentials(req.body);
```

```
//  /* Caso ocorra erro na validação do Joi, passamos esse */

  // ...
```

8. O próximo passo é criar o controller para nossa nova rota. Crie o arquivo `controllers/signup.js` :

Copiar

```
const { validateCredentials } =
require('./utils/validateCredentials');

const service = require('../services/User');

module.exports = async (req, res, next) => {
  /* Começamos validando `username` e `password`  */
  const { error: validationError } = validateCredentials(req.body);

  /* Caso haja erro de validação */
  if (validationError) {
    /* Enviamos o erro para o middleware de erro */
    return next(validationError);
  }

  const { username, password } = req.body;

  /* Pedimos para o service armazenar os dados */
  /* Em troca, recebemos o resultado da ação */
  const result = await service.create(username, password);

  /* Validamos se o retorno tem algum erro */
  /* Se não tiver, retornamos o token*/
  if (!result.error) {
    return res.status(201).json(result);
  }

  /* Se tiver algum erro, e o código for "usernameExists" */
  /* Retornamos um 409 Conflict com a mensagem do erro */
  if (result.error.code === 'usernameExists') {
    return res.status(409).json({ message: result.error.message });
  }
};
```

9. Agora, adicione o arquivo ao index de controllers. Edite o arquivo `controllers/index.js` :

Copiar

```
// const me = require('./me');
// const ping = require('./ping');
// const login = require('./login');
// const topSecret = require('./topSecret');
const signup = require('./signup');

// module.exports = {
//   me,
//   ping,
//   login,
//   topSecret,
  signup,
// };
```

10. E, por fim, adicione o endpoint ao express. Edite o arquivo `index.js` :

Copiar

```
// ...
// app.get(
//   '/top-secret',
//   /* Middleware que valida o JWT e cria `req.user` */
//   middlewares.auth,
//   /* Middleware que verifica se a pessoa autenticada é admin */
//   middlewares.admin,
//   /* Controller do endpoint */
//   controllers.topSecret,
// );

app.post('/signup', controllers.signup);

// app.use(middlewares.error);
// ...
```

Exercício 2

2. Altere o endpoint de login
3. Antes de gerar o token, verifique se o nome de usuário e a senha informados existem no arquivo `users.json` ;

4. Não permita mais o login do usuário `admin` com a senha fixa.
5. Informe, na propriedade `admin` do payload do token, o mesmo valor da propriedade `admin` que está armazenado para aquela pessoa.

Resolução

1. Quem realiza a autenticação é, na verdade, o service `User` . Sendo assim, é ele quem precisamos alterar. Altere o arquivo `services/User.js` :

Copiar

```
// const jwt = require('jsonwebtoken');
// const model = require('../models/User');

// const { JWT_SECRET } = process.env;

/* Deixamos de receber `admin`, pois agora será lido de Users.json
*/
 const login = async (username, password) => {
//   /* Não precisamos validar os campos, pois o controler já faz
isso pra nós */

  /* Buscamos as informações no arquivo Users.json */
  const user = await model.findOne(username);

  if (!user || user.password !== password) {
    return {
      error: {
        code: 'invalidCredentials',
        message: 'Invalid username or password',
      },
    };
  }

  const payload = {
    username,
    /* Usamos a informaçào no arquivo Users.json para determinar
       se a pessoa é admin */
    admin: user.admin,
  };

//   const token = jwt.sign(payload, JWT_SECRET, {
//     expiresIn: '1h',
```

```
//   });

//   return { token };
// };

// const create = async (username, password) => {
//   /* A primeira coisa que precisamos fazer
//   é verificar se o username informado já existe */
//   const userExists = await model.findOne(username);

//   /* Caso o username já exista */
//   if (userExists) {
//     /* Retornamos um objeto de erro */
//     return {
//       error: {
//         message: 'Username already exists',
//         code: 'usernameExists',
//       },
//     };
//   }

//   /* Caso o username não exista, "rolamos o dado" para descobrir
se essa pessoa será admin */
//   const admin = Math.floor(Math.random() * 100) > 50;

//   /* Depois, armazenamos os dados no arquivo */
//   await model.create({ username, password, admin });

//   /* Por fim, retornamos o token  */
  /* NÃo precisamos mais passar o valor de admin, pois será lido do
arquivo */
  return login(username, password);
// };

// module.exports = {
//   create,
//   login,
// };
```