10

Autenticação baseada em token

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/mean-js/stages/12-alurapic.zip) completo do projeto do capítulo anterior e continuar seus estudos a partir deste capítulo. Só não esqueça de baixar as dependências do projeto no terminal com o comando `npm install`.

JWT (JSON Web Token)

Hoje qualquer um que saiba a URL do nossa aplicação é capaz de incluir fotos. A ideia agora é que apenas usuários autenticados possam realizar essa tarefa. Para isso, usaremos um sistema de autenticação baseado em tokens usando o padrão JWT. O JSON Web Token (JWT) é um padrão aberto que define um meio de transmitir informações entre interessados utilizando JSON object. A informação é confiável porque é assinada digitalmente.

Precisamos realizar uma série de modificações em nosso projeto, inclusive em nosso cliente Angular para conseguirmos ativar nosso sistema de autenticação. Focarei no fundamental sem pensar em todos os casos extremos para não ofuscar seu entendimento.

O modelo de Usuário

O primeiro passo será criarmos um novo modelo que representa o usuário do nosso sistema. Ele terá apenas `login` e `senha`. Criando o arquivo `alurapic/app/models/usuario.js`:

```
// alurapic/app/models/usuarios.js

var mongoose = require('mongoose');

var schema = mongoose.Schema({

    login: {
        type: String,
        required: true
    },
    senha: {
        type: String,
        required: true
    }
});

mongoose.model('Usuario', schema);
```

É através deste model que buscaremos o usuário no banco durante o processo de autenticação. É claro, precisamos de pelo menos um usuário cadastrado. Vou adicionar o novo usuário diretamente no MongoDB:

```
MongoDB shell version: 3.0.7
connecting to: test
> use alurapic
switched to db alurapic
> db.usuarios.insert({login: 'flavio', senha: '123'});
```

A parcial de login

O próximo passo será criarmos uma parcial de login em nosso cliente Angular. Não custa nada eu lembrar que se você não fez o treinamento de Angular do Alura, não fará ideia dos arquivos que eu estou alterado. Aliás, tudo que farei inicialmente já foi ensinado neste treinamento. Darei um destaque especial para as novidades que aparecerem.

Muito bem, vamos criar **alurapic/public/partials/login.html**:

```
<h1 class="text-center">Login</h1>

<p class="alert-danger"></p>
<form>
    <div class="form-group">
        <label>Login</label>
        <input type="text" class="form-control">
    </div>
    <div class="form-group">
        <label>Senha</label>
        <input type="password" class="form-control">
    </div>
    <input type="submit" value="Entrar" class="btn btn-primary">
</form>
```

É claro, para acessarmos essa parcial, precisamos de uma rota em nossa aplicação Angular. Vamos editar **public/js/main.js**:

```
angular.module('alurapic', ['minhasDiretivas','ngAnimate', 'ngRoute', 'ngResource', 'meusServicos'])
    .config(function($routeProvider, $locationProvider, $httpProvider) {

         // código anterior omitido

        $routeProvider.when('/login', {
            templateUrl: 'partials/login.html',
            controller: 'LoginController'
        });

        $routeProvider.otherwise({redirectTo: '/fotos'});
    });
```

Criamos a rota `/login` que carregará `login.html`, perfeito. Note que a parcial está associada com o controller `LoginController` que ainda não existe. Vamos criar o arquivo **alurapic/public/js/controllers/login-controller.js**:

```
angular.module('alurapic').controller('LoginController', function($scope, $http, $location) {

    $scope.usuario = {};
    $scope.mensagem = '';

    $scope.autenticar = function() {

        var usuario = $scope.usuario;

        $http.post('/autenticar', {login: usuario.login, senha: usuario.senha})
        .then(function() {
```

```
                $location.path('/');
            }, function(erro) {
                $scope.usuario = {};
                $scope.mensagem = 'Login/Senha incorretos';
            });
        };
    });
```

Quando a função `$scope.autenticar` for executada, ela enviará um JSON com o login e a senha do usuário para o servidor. Sabemos que no servidor, ainda não há a rota `/autenticar`. Daqui a pouco implementarei essa rota.

Lembre que isso não é suficiente, é necessário importar o script em `alurapic/public/index.html`:

```
<!-- alurapic/public/index.html -->
<head>
<!-- outros scripts -->
<script src="js/controllers/login-controller.js"></script>
</head>
```

Isso já deve ser suficiente para conseguirmos visualizar a página de login através do endereço `localhost:3000/#/login`. Formulário sendo exibido? Podemos continuar!

Bom, precisamos alterar a parcial `alurapic/public/partials/login.html` para chamar `$scope.autenticar` de `LoginController`, inclusive associar os campo de login e senha através de `ng-model`. Também associaremos através de angular expression um parágrafo que exibirá a mensagem de login inválido.

```
<!-- alurapic/public/partials/login.html -->

<h1 class="text-center">Login</h1>

<p class="alert-danger">{{mensagem}}</p>
<form ng-submit="autenticar()">
    <div class="form-group">
        <label>Login</label>
        <input type="text" ng-model="usuario.login" class="form-control">
    </div>
    <div class="form-group">
        <label>Senha</label>
        <input type="password" ng-model="usuario.senha" class="form-control">
    </div>
    <input type="submit" value="Entrar" class="btn btn-primary">
</form>
```

Pronto, primeira de muitas partes pronta. Agora vamos implementar a rota `/autenticar` lá em nosso backend Node.

Rotas de autenticação e segurança

Vamos criar o arquivo `alurapic/app/routes/auth.js`. Todas as rotas envolvidas na autenticação serão adicionadas neste arquivo:

```
// alurapic/app/routes/auth.js

module.exports = function(app) {

    var api = app.api.auth;
    app.post('/autenticar', api.autentica);
    app.use('/*', api.verificaToken);
};
```

Teremos duas rotas. A primeira `/autenticar` não é protegida e nem poderia, porque é aquela que será usada pela aplicação Angular não autenticada para se autenticar. Autenticação bem sucedida devolve para aplicação Angular um token que deve ser armazenado e posteriormente enviado à cada requisição. Entenda-o como uma credencial. Todas os recursos que acessarmos em nosso servidor, exceto a rota `/autenticar`, precisará dessa credencial.

A rota `/*` é especial. Primeiro, porque usamos `app.use`. Isso permite respondermos a qualquer verbo HTTP. Esta rota é aquela disparada independente da URL acessada pelo usuário e que verificará se há um token (credencial) na requisição, inclusive se ele é válido. Caso alguém tente acessar um recurso sem enviar um token (aquele obtido na autenticação) ou um token que não é mais válido (que expirou) o servidor devolverá o código de status 401. Como a rota `/autenticar` foi definida antes de `/*`, a primeira não será filtrada pela segunda. Perfeito, pois não queremos bloquear acesso à rota `/autenticar`.

Bom, estamos acessando `app.api.auth` que ainda não existe em nossa api. Vamos criá-lo:

```
// alurapic/app/api/auth.js

var mongoose = require('mongoose');

module.exports = function(app) {

     var api = {};
     var model = mongoose.model('Usuario');

     api.autentica = function(req, res) {

         model.findOne({
             login: req.body.login,
             senha: req.body.senha
         })
         .then(function(usuario) {
             if (!usuario) {
                 console.log('Login/senha inválidos');
                 res.sendStatus(401);
             } else {
                 // elabora o token e envia para aplicação Angular
             }
         });
     };

     api.verificaToken = function(req, res, next) {
     };

    return api;
};
```

Criando um JWT

Em `api.autentica`, usamos nosso modelo do Mongoose para buscar o usuário pelo seu login e senha em nosso banco. Caso não exista, enviamos o código 401. E se existir? Precisamos elaborar como resposta nosso token. Usaremos o módulo [jsonwebtoken (https://github.com/auth0/node-jsonwebtoken)](https://github.com/auth0/node-jsonwebtoken) para essa finalidade. Instalando-o através do terminal:

```
npm install jsonwebtoken@5.4.1 --save
```

Ótimo, já temos nosso módulo baixado, mas você precisa saber que precisaremos de uma senha, um segredo que será usado para criptografar o token. Vamos definir essa configuração em `alurapic/config/express.js` como uma variável de ambiente, mas antes de chamarmos o módulo `consign`:

```
// alurapic/config/express.js

// código anterior omitido
app.set('secret', 'homemavestruz');
// código posterior omitido
```

Eu coloquei uma frase qualquer, invente a sua! Voltando para nosso arquivo `alurapic/app/api/auth.js`:

```
// alurapic/app/api/auth.js

var mongoose = require('mongoose');
var jwt  = require('jsonwebtoken'); // importando o módulo

module.exports = function(app) {

     var api = {};
     var model = mongoose.model('Usuario');

     api.autentica = function(req, res) {

         model.findOne({
             login: req.body.login,
             senha: req.body.senha
         })
         .then(function(usuario) {
             if (!usuario) {
                 console.log('Login/senha inválidos');
                 res.sendStatus(401);
             } else {
                 var token = jwt.sign( {login: usuario.login}, app.get('secret'), {
                     expiresIn: 86400 // valor em segundo, aqui temos um total de 24 horas
                 });
                 console.log('Autenticado: token adicionado na resposta');
                 res.set('x-access-token', token); // adicionando token no cabeçalho de resposta
                 res.end(); // enviando a resposta
             }
         });
     };

     api.verificaToken = function(req, res, next) {
    };
```

```
    return api;
};
```

Vejamos o código:

```
var token = jwt.sign( {login: usuario.login}, app.get('secret'), {
    expiresIn: 86400 // valor em segundo, aqui temos um total de 24 horas
});
```

Neste techo, chamamos a função `jwt.sign` passando como primeiro parâmetro um *payload*. Um *payload* é qualquer valor escolhido por nós, no caso da aplicação, estou passando o login do usuário. O segundo parâmetro estamos solicitando à variável de ambiente do Express `secret` a senha armazenada. É essa senha que será usada para assinar o token (não preciso dizer o quanto ela é importante, não?). O terceiro e último parâmetro é um JSON que permite passar algumas configurações para o módulo. No meu exemplo, uso a propriedade `expiresin` para indicar quando o token espirará. Ele aceita um valor em segundo, por isso passei 86400 para o token expirar em 24h.

Por fim, temos:

```
res.set('x-access-token', token);
res.end()
```

Neste trecho, estou adicionando um novo valor ao cabeçalho da resposta que estamos para enviar. No caso, usei a chave `x-access-token` , mas poderia ser qualquer nome. Seu valor é o token que acabamos de criar. Em seguida, chamamos `res.end()` para enviar a resposta para o cliente. Fique atento, porque neste momento devolvemos a resposta para um login bem sucedido e a aplicação angular terá que extrair desse header o token e armazená-lo para enviá-lo sempre nas requisições posteriores.

Filtrando requisições

Agora, vamos pular para aquele nosso filtro, que barra as requisições que não tiverem enviado um token valido:

```
// alurapic/app/api/auth.js

api.verificaToken = function(req, res, next) {

        var token = req.headers['x-access-token']; // busca o token no header da requisição

        if (token) {
            console.log('Token recebido, decodificando');
            jwt.verify(token, app.get('secret'), function(err, decoded) {
                if (err) {
                    console.log('Token rejeitado');
                    return res.sendStatus(401);
                } else {
                    console.log('Token aceito')
                    // guardou o valor decodificado do token na requisição. No caso, o login do usu
                    req.usuario = decoded;
                    next();
                }
            });
```

```
        } else {
            console.log('Nenhum token enviado');
            return res.sendStatus(401);
          }
    }
```

É através de `jwt.verify` que verificamos se o token que extraímos da requisição é válido. Se não for, enviamos `401` , se for, guardamos o valor do token decodificado na requisição, no caso, o login do usuário. Só guardei na requisição esta informação porque pode ser útil, caso alguém queira saber o nome do usuário logado no sistema. A grande jogada está na chamada da função `next()` . Só temos acesso à esta função, porque definimos nossa rota com `app.use` . A função `next()` passa o controle da requisição para os próximos middlewares da pilha. É claro, todos esses outros middlewares (inclusive todas as outras rotas) terão acesso à `req.usuario` .

Obs.: É importante salientar que neste exemplo a variável `req.usuario` não foi usada e foi incluída para indicar uma alternativa de uso na aplicacão.

Ajustando o consign

Só um detalhe importante na organização da nossa MEAN Stack. Aprendemos que o `consign` carregará todas as nossas rotas em `alurapic/app/routes` , inclusive nossos modelos e API. A questão toda é que `alurapic/app/routes/auth.js` precisa ser a primeira rota a ser carregada. O consign lê os arquivo em ordem alfabética e por sorte nossa rota será a primeira a ser carregada. Mas se quiséssemos usar outro nome, por exemplo, `usuario-autentica.js` ? Com certeza teríamos problema, porque nosso filtro que barra as requisições sem token só seria processado depois de terem acessado nossas rotas.

Para resolver isso, podemos indicar para o consign que carregue um módulo específico antes dos outros. Alterando `alurapic/config/expres.js` :

```
// alurapic/config/expres.js


consign({cwd: 'app'})
    .include('models')
    .then('api')
    .then('routes/auth.js')
    .then('routes')
    .into(app);
```

Veja que antes de carregamos a pasta `routes` , primeiro carregamos `routes/auth.js` garantindo assim que nossa rota será carregada primeiro.

Vamos reiniciar nosso servidor e tentar acessar nossa aplicação. Nenhum foto será exibida e se abrirmos o terminal do Chrome veremos vários código `401` .

Angular e interceptadores

Para completar a funcionalidade precisamos alterar nossa aplicação Angular mais uma vez. Angular permite trabalhar com interceptadores de requisições Ajax. Esse recurso me permitirá escrutinar qualquer requisição, inclusive qualquer resposta do servidor. Implementaremos a seguinte lógica nesse interceptador:

1 - Toda resposta recebida procuraremos no header se é o token `x-access-token` . Se existir, guardaremos esse token no sessionStorage do navegador. O token ficará até que o navegador seja fechado. Vale lembrar que apenas durante uma autenticação bem sucedida o token será enviado.

2 - A cada requisição feita em nossa aplicação, verificaremos se existe um token armazenado no sessionStorage do browser, se existir, o token será adicionado no header da requisição.

3 - Recebemos o código 401 quando o usuário acessar um recurso sem ter enviado um token, ou seu token expirou e ele não foi autorizado. Nessa situação removemos qualquer token que esteja no sessionStorage (pode ser inválido) e faremos um redirecionamento no lado do cliente Angular para a parcial de login. Isso significa que se ele tentar acessar qualquer parte da nossa aplicação que realize uma requisição Ajax ele será redirecionado para a página de login.

Voltando ao nosso interceptador. Um interceptador é um serviço com a diferença de que é ativado de uma maneira especial e possui uma estrutura padrão. Vamos criar o serviço, nosso interceptador no arquivo `alurapic/public/js/service/token-inteceptor.js` :

```
angular.module('alurapic')
    .factory('tokenInterceptor', function($q, $window, $location) {

        var interceptor = {};

        interceptor.request = function(config) {
            // enviar o token na requisição
            config.headers = config.headers || {};
            if ($window.sessionStorage.token) {
                console.log('Enviando token já obtido em cada requisição');
                config.headers['x-access-token'] = $window.sessionStorage.token;
            }
            return config;
        },

        interceptor.response = function (response) {
            var token = response.headers('x-access-token');
            if (token != null) {
                $window.sessionStorage.token = token;
                console.log('Token no session storage: ', token);
            }
            return response;
        },

        interceptor.responseError = function(rejection) {

            if (rejection != null && rejection.status === 401) {
                console.log('Removendo token da sessão')
                delete $window.sessionStorage.token;
                $location.path("/login");
            }
            return $q.reject(rejection);
        }

    return interceptor;

});
```

Nosso serviço `tokenInterceptor` possui as propriedades `request` , `response` e `responseError` e cada uma delas armazena uma função. Esses nomes não foram escolhidos por acaso, é uma exigência quando estamos criando um interceptador.

A função atribuída à `response` será chamada sempre que recebermos uma resposta do servidor usando `$http` ou `$resource` . É o momento ideal para verificarmos se um token foi enviado pelo servidor. Sabemos que ele só será enviado na reposta de uma autenticação de sucesso, fora desse cenário, ele sempre será nulo. É por isso que só tomamos uma ação quando o token está presente. Usamos o serviço `$windows` para podermos acessar o `sessionStorage` do navegador no estilo Angular. Com o token em mãos, adicionamos a propriedade `$windows.sessionStorage.token` dinamicamente adicionando o token recebido.No final precisamos retornar a resposta ou uma promise.

A função atribuída à `request` executada toda vez que uma requisição for realizada, ou seja, antes da requisição ser efetivada. Nela, verificamos se há um token em `$windows.sessionStotrage.token` . Se houver, adicionamos esse token no header de requisição garantindo seu envio para o servidor à cada requisição.

Por fim, em `responseError` , a função só será executada quando recebermos um código de status indicando um erro na requisição. Não estamos interessados em qualquer erro, mas no 401. Quando acontecer, é porque o usuário tentou acessar um recurso sem ter enviado um token ou com um token inválido. É por isso que apagamos um possível token inválido do sessionStorage e em seguida redirecionamentos o usuário via rota do Angular para a tela de login.

É claro, precisamos importar nosso interceptador em `alurapic/public/index.html` :

```
<script src="js/services/token-interceptor.js"></script>
```

E agora mais uma novidade. Precisamos ativar esse interceptador através do `$httpProvider` , um artefato injetável na função `.config` lá da configurações das nossas rotas.

```
angular.module('alurapic', ['minhasDiretivas','ngAnimate', 'ngRoute', 'ngResource', 'meusServicos'])
    .config(function($routeProvider, $locationProvider, $httpProvider) {

        $httpProvider.interceptors.push('tokenInterceptor');

       // código de configuração de rotas omitido
```

Cuidado, pois a string que você passar para a função `push` tem que ter exatamente o mesmo nome do nosso interceptador.

Pronto, com isso temos todas as pessoas encaixadas. Se tentarmos acessar o sistema, seremos redirecionamos para a tela de login. Enquanto não nos autenticarmos, sempre seremos jogados para essa página. Se a autenticação for bem sucedida, seremos lançados para a página principal da aplicação.

E se quisermos nos deslogar? Basta criar um controller que delete o token do session storage, por exemplo:

```
delete $window.sessionStorage.token
```

Isso será suficiente para direcionar o usuário para a página de login. As possibilidades são muitas.