05

Lapidando o lapidado

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/mean-js/stages/04-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` .

Configuramos a rota `v1/fotos` através do Express disponibilizando uma lista de fotos para nossa aplicação Angular. Muito bem, mas durante o treinamento de Angular requisitávamos outros recursos do servidor, como uma lista de **grupos** para popular a combobox do cadastro de fotos. Cedo ou tarde teremos que criar uma rota para este recurso, que tal criá-la agora?

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

var express = require('express');
var app = express();

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

app.get('/v1/fotos', function(req, res) {

    var fotos = [
        {_id: 1, titulo: 'Leão', url:'http://www.fundosanimais.com/Minis/leoes.jpg' },
        {_id: 2, titulo: 'Leão 2', url:'http://www.fundosanimais.com/Minis/leoes.jpg' }
    ];

    res.json(fotos);

});

// nova rota!
app.get('/v1/grupos', function(req, res) {

    var grupos = [
        { _id: 1, nome: 'esporte' },
        { _id: 2, nome: 'lugares' },
        { _id: 3, nome: 'animais' }
    ];

    res.json(grupos);
});

module.exports = app;
```

O código é semelhante ao que fizemos anteriormente. Para testarmos, basta iniciarmos o servidor e acessarmos o endereço `localhost:3000/v1/grupos` . Ótimo, tudo funciona e toda vez que precisamos criar uma nova rota basta adicioná-la no arquivo `alurapic/config/express.js` , certo? E se pudermos fazer melhor?

Separando o ouro da rocha, mas preservando a natureza

Que tal isolarmos cada configuração de rota em seu próprio arquivo? Isso facilitará em muito a manutenção, porque quando darmos manutenção no código de uma rota específica, abriremos apenas o arquivo desta rota. E mais do que isso, se por

acaso eu cometesse um erro no arquivo, só comprometeria o código daquela rota e deixaria intacto os demais arquivos. Que tal?

Para isso, criei a pasta `alurapic/app`. Nesta pasta ficará todo o código da aplicação, aquele que não tem relação com configuração ou infraestrutura. Isso ajudará muito a mim e a outros desenvolves em compreender o que a aplicação faz, sem ter que esbarrarmos com arquivos de configurações ou código de infraestrutura. Tudo bem que teremos a configuração das rotas dentro dessa pasta, mas essas configurações são de alto nível, porque indicam o que nossa aplicação é capaz de realizar, não é um código que configura o Express ou o banco de dados, algo que eu quero configurar uma vez e esquecer.

Em seguida, criarei a subpasta `alurapic/app/routes`, aquela que conterá o arquivo de rotas do sistema. Vou criar o arquivo `foto.js` e para dentro dele moverei o código que configura as duas rotas da nossa aplicação.

```
// alurapic/app/routes

app.get('/v1/fotos', function(req, res) {

    var fotos = [
        {_id: 1, titulo: 'Leão', url:'http://www.fundosanimais.com/Minis/leoes.jpg' },
        {_id: 2, titulo: 'Leão 2', url:'http://www.fundosanimais.com/Minis/leoes.jpg' }
    ];

    res.json(fotos);

});

app.get('/v1/grupos', function(req, res) {

    var grupos = [
        { _id: 1, nome: 'esporte' },
        { _id: 2, nome: 'lugares' },
        { _id: 3, nome: 'animais' }
    ];

    res.json(grupos)
});
```

Claro, como tiramos esse código de `alurapic/config/express.js`, nosso arquivo final fica assim:

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

var express = require('express');
var app = express();

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

// configuração de rotas foi movido para o arquivo alurapic/app/routes/foto.js
module.exports = app;
```

Separar, não isolar

Vamos testar? Não funciona, o nosso servidor parece não estar ciente das rotas que foram isoladas. Claro, em nenhum momento pedimos para ele carregar o arquivo de rota. Vamos alterar `alurapic/config/express.js`:

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

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

require('../app/routes/foto');

module.exports = app;
```

Quando testamos, nosso servidor explode com a mensagem de erro:

```
ReferenceError: app is not defined
```

Faz sentido, porque dentro de `alurapic/app/routes/foto.js` estamos querendo acessar uma instância do Express através da variável `app` que não existe dentro do módulo. Não podemos simplesmente criar uma nova instância do Express, por esta não estaria configurada, por exemplo, com o middleware que compartilha nossa pasta `public` . Uma solução é fazer com que nosso módulo receba como parâmetro a mesma instância criada em `alurapic/config/express.js` :

```
module.exports = function(app) {

    app.get('/v1/fotos', function(req, res) {

        var fotos = [
            {_id: 1, titulo: 'Leão', url:'http://www.fundosanimais.com/Minis/leoes.jpg' },
            {_id: 2, titulo: 'Leão 2', url:'http://www.fundosanimais.com/Minis/leoes.jpg' }
        ];

        res.json(fotos);

    });

    app.get('/v1/grupos', function(req, res) {

        var grupos = [
            { _id: 1, nome: 'esporte' },
            { _id: 2, nome: 'lugares' },
            { _id: 3, nome: 'animais' }
        ];

        res.json(grupos)
    });
};
```

Veja que envolvemos todo nosso código dentro da função:

```
module.exports = function(app) {
  // nosso código
};
```

Com isso, não estamos mais importando um objeto, mas uma função! Além disso, essa função recebe como parâmetro uma instância do Express. Vamos testar?

Durante meu teste, nada acontece. Parece que o código dentro do nosso novo módulo não é executado. Mas é claro que não funcionará, como agora nosso módulo é uma função, em nosso módulo `alurapic/config/express.js` não basta realizar `require('../app/routes/foto')`. Como nosso módulo agora retorna uma função, precisamos executá-la:

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

var express = require('express');
var app = express();

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

// fazendo o require e invocando a função com ()
require('../app/routes/foto')();

module.exports = app;
```

Será que funciona? Vamos reiniciar o servidor e tentar novamente. Não funciona, e ainda recebemos uma mensagem de erro no servidor que nem inicia:

```
TypeError: Cannot read property 'get' of undefined
```

Separe, mas o conjunto precisa continuar a funcionar

E não vai funcionar mesmo! Nosso módulo é uma função que espera receber uma instância configurada do Express e em nenhum momento passamos essa instância! Vamos corrigir:

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

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

// passando a instância configurada do Express
require('../app/routes/foto')(app);

module.exports = app;
```

Agora sim! Reiniciando o servidor e testando nosso server continua funcionando. Podemos continuar e aprender mais coisas? Que tal um pouco mais de organização?

Paradoxo: separando criamos uma unidade mais forte

Bom, vamos dar uma olhadinha em nosso arquivo `alurapic/app/routes/foto.js`. Nele temos duas rotas, uma para recurso de fotos e outra para recurso de grupos. Vamos separar a do grupo também? Vamos mover o código para seu próprio arquivo `alurapic/app/routes/grupos`.

```
// alurapic/app/routes/grupo.js

module.exports = function(app) {
```

```
    app.get('/v1/grupos', function(req, res) {

        var grupos = [
            { _id: 1, nome: 'esporte' },
            { _id: 2, nome: 'lugares' },
            { _id: 3, nome: 'animais' }
        ];

        res.json(grupos)
    });
};
```

É claro, nosso arquivo `alurapic/app/routes/foto.js` ficou assim:

```
module.exports = function(app) {

    app.get('/v1/fotos', function(req, res) {

        var fotos = [
            {_id: 1, titulo: 'Leão', url:'http://www.fundosanimais.com/Minis/leoes.jpg' },
            {_id: 2, titulo: 'Leão 2', url:'http://www.fundosanimais.com/Minis/leoes.jpg' }
        ];

        res.json(fotos);

    });

};
```

Vamos testar reiniciar e testar nosso servidor. Primeiro, vamos acessar:

```
http://localhost:3000/v1/fotos
```

Perfeito, a lista de fotos é retornada. Agora vamos testar:

```
http://localhost:3000/v1/grupos
```

Opa, esse não funciona, parece que não foi configurado. Claro, esquecemos de importar o módulo `alurapic/app/routes/grupo.js` em nosso `express.js`:

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

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

require('../app/routes/foto')(app);
require('../app/routes/grupo')(app); // importação que faltava

module.exports = app;
```

Agora sim, está funcionando.

Separação manual, união automática

Toda vez que criarmos um novo arquivo de rota, basta importarmos esse arquivo através da função `require` em `alurapic/config/expres.js`. Mas e se o programador esquecer como eu esqueci? A aplicação funcionará? Aliás, você lembrará de adicionar essa configuração? É por isso que usaremos um módulo especial que carregará automaticamente todos nossos arquivos de rota. Se tivermos três arquivos, serão carregados os três. Se adicionarmos mais um, ele também será carregado, tudo sem precisar alterar `alurapic/config/express.js`.

Vamos instalar através do npm o módulo consign (https://github.com/jarradseers/consign) que fará essa mágica para nós:

```
npm install consign@0.1.2 --save
```

Ótimo, baixamos o módulo consign que está armazenado em `alurapic/node_modules` e que podemos importar através função `require`. Como usá-lo? Primeiro, vamos abrir o arquivo `alurapic/config/expres.js` para importar o módulo e em seguida remover as duas funções `require` que importam nossos módulos de rotas:

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

var express = require('express');
var consign = require('consign'); // importação do módulo mágico!
var app = express();

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

// requires removidos

module.exports = app;
```

Agora, vamos usar o módulo para incluir nosso diretório `alurapic/app/routes`. O diretório? Sim, porque todos os módulos que tiverem dentro desta pasta serão carregados pelo `consign` automaticamente. Modificando nosso arquivo:

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

var express = require('express');
var consign = require('consign');
var app = express();

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

// incluindo a pasta `app/routes`.
consign().include('app/routes');

module.exports = app;
```

Veja que passamos como parâmetro `app/routes` e não `../app/routes`. A razão disso é que o consign considera como raiz a pasta na qual nossa aplicação foi iniciada. Como iniciamos a aplicação executando `alurapic/server.js`, a pasta raiz é `alurapic`. Se o consign considerasse a pasta o local onde o arquivo `alurapic/config/express.js` está, teríamos que descer uma pasta com a sintaxe `../app/routes`.

Chega de papo, funciona ou não? Só reiniciando nosso servidor e testando. Pelo menos o servidor sobre e ainda mostra no terminal que carregou nossos arquivos:

```
consign v0.1.2 Initialized in alurapic
+ ./app/routes/foto.js
+ ./app/routes/grupo.js
```

Ótimo! Mas um teste demonstra que não funciona. Nem deveria, porque em que momento o `consign` sabe que deve passar para esses módulos a instância do Express que precisam para funcionar? Fazemos isso com a função `into`:

```
var express = require('express');
var consign = require('consign');
var app = express();

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

// carrega todos os arquivo dentro de app/routes e passa app como parâmetro para eles
consign().include('app/routes').into(app);

module.exports = app;
```

A função `.into` recebe nossa instância do Express que será passada para todos esses módulos. Testando mais uma vez... ótimo, tudo continua funcionando! Quer dizer que se eu criar um novo arquivo dentro de `alurapic/app/routes ele será carregado automaticamente? Sim, mas apenas quando o servidor reiniciar, não com ele rodando, tá? Ainda assim, isso evita que eu tenha que lembrar de alterar o nosso arquivo de configuração do Express, perfeito!

Flávio, e agora? Podemos continuar? Só mais uma alteração. Temos dois arquivos de rotas, certo? Mas se olharmos o arquivo `alurapic/app/routes/foto.js` temos nele a rota e a lógica que será executada quando ela for acessada. Temos dois motivos para alterar esse arquivo, um quando o endereço da rota mudar e outro quando a lógica executada também mudar. É por isso que vou deixar a rota onde está, mas o código que será executado em outro arquivo dentro de `alurapic/app/api`. Por que esse nome?

Quando trabalhamos com SPA (nosso framework Angular) o que o servidor fornece para a aplicação que roda no browser é um conjunto de API's (application program interface) que pode ser consumido. Essas API's recebem e retornam dados apenas, não há qualquer lógica de visão ou página gerada dinamicamente. Isso difere um pouco do modelo tradicional quando o browser pede algo ao servidor e este retorna uma página já construída e mesclada com dados. Quem é o responsável em obter dados e construir a view dinamicamente é nossa aplicação Angular, não o server. A mesma API pode alimentar uma aplicação feita em Android, jQuery ou qualquer outra tecnologia que trabalhe com http.

Voltando... A pasta API será aquela que terá toda a nossa lógica, mas desprovida de um endereço. Inclusive podemos ter lógica acessada por rotas diferentes. Este cenário ocorre quando desejamos mudar endereço da rota, mas ainda assim queremos manter o endereço antigo para não quebrar outras aplicações que acessando o endereço antigo e que não foram atualizadas.

Com a pasta `alurapic/app/api` criada, vamos criar dois arquivos: `foto.js` e `grupo.js` dentro dessa pasta. Se você reparar, os arquivos em o mesmo nome dos arquivo criados em `alurapic/app/routes`. Utilizei o mesmo nome, mas nada o impede de usar qualquer outro.

Vou começar por `alurapic/app/api/foto.js`. Este módulo precisa retornar um objeto Javascript com propriedades que guardam nossa lógica, inclusive podemos dar nomes bem interessantes para essas propriedades.

```
var api = {}; // objeto

// adiciona dinamicamente a propriedade `lista` associando uma função
api.lista = function(req, res) {

    // lógica aqui
};

module.exports = api;
```

Criamos um objeto que possui a propriedade `lista`, aquela que será chamada quando quisermos listar todas as fotos. Podemos ter `adiciona`, `remove`, `altera` e por aí vai. Agora, em `alurapic/app/routes/fotos.js`, vamos copiar na íntegra o conteúdo da função passada como segundo parâmetro da função `app.get`. Nosso arquivo `alurapic/app/api/fotos.js` fica assim:

```
var api = {};

api.lista = function(req, res) {

    var fotos = [
        {_id: 1, titulo: 'Leão', url:'http://www.fundosanimais.com/Minis/leoes.jpg' },
        {_id: 2, titulo: 'Leão 2', url:'http://www.fundosanimais.com/Minis/leoes.jpg' }
    ];

    res.json(fotos);
};

module.exports = api;
```

Veja que temos código duplicado em `alurapic/app/api/foto.js` e também em `alurapic/routes/foto.js`. A ideia é usarmos o código da nossa API em nosso arquivo de rota. Vamos alterar o arquivo de rota:

```
// alurapic/app/routes

// importa nosso arquivo com a API de fotos
var api = require('../api/foto');

module.exports = function(app) {
       // passa a função api.lista como parâmetro
    app.get('/v1/fotos', api.lista);

};
```

Funciona, mas podemos evitar de fazer o `require` da nossa api em nosso arquivo de rotas. Como? No `consign` precisamos carregar **primeiro** a pasta `api` e logo em seguida a `routes`:

```
// alurapic/config/express.js
var express = require('express');
var consign = require('consign');
var app = express();
```

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

consign()
    .include('app/api')
    .then('app/routes')
    .into(app);

module.exports = app;
```

No consign, a primeira pasta carregada usamos a função `include`, para as outras, usamos a função `then`. Então, Fizemos primeiro o include de `app/api` e depois, através da função `then`, `app/routes`. Isso não é por acaso. Se nossas rotas dependem da API para funcionar, a API deve ser carregada primeiro. Vou explicar.

Em nosso projeto, o script alurapic/app/api/foto.js se torna disponível através da instância do Express como **app.api.foto**. Quando criamos o `alurapic/app/api/grupo.js` ele estará disponível como **app.api.grupo**. O que o consign faz é pendurar na instância do Express nossas API's usando a mesma hierarquia de diretórios e nome de arquivos. Vamos alterar `alurapic/app/routes/fotos.js` para substituirmos a função `require` pelo acesso da nossa API diretamente da instância do Express:

```
// alurapic/app/routes

// removi o require!

module.exports = function(app) {

      // jogando para uma variável para deixar mais claro
    var api = app.api.foto;

    app.get('/v1/fotos', api.lista);
};
```

Funciona? Sou igual a São Tomé, só acredito vendo: vamos reiniciar o servidor e verificar o resultado. Um erro!

```
/Users/flavioalmeida/Desktop/alurapic/app/routes/foto.js:5
    var api = app.api.foto;
                      ^

TypeError: Cannot read property 'foto' of undefined
```

Hum, já sei o problema, Vamos ver a configuração do `consign`:

```
consign()
    .include('app/api')
    .then('app/routes')
    .into(app);
```

Como usamos `app/api` e `app/routes`, pra funcionar seria preciso (você não vai alterar, ok?) alterar para:

```
app.app.api.foto
```

Mas fica estranho termos duas `app`, não? Podemos resolver isso dizendo para o `consign` qual o diretório de trabalho atual (current working directoty CWD) que ele buscará `api` e `routes`. Isso fará que esse diretório padrão não entre na hierarquia das propriedades dos objetos. Vamos alterar `alurapic/config/express.js`:

```
var express = require('express');
var consign = require('consign');
var app = express();

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

consign({cwd: 'app'})
    .include('api')
    .then('routes')
    .into(app);

module.exports = app;
```

Testando... funcionou!

E que tal realizarmos a separação da lógica em `alurapic/app/routes/grupo.js` em seu próprio arquivo? Mas isso será um dos nossos exercícios.

Aprendemos neste capítulo a separar a configuração das rotas do arquivo de configuração do Express, tornando mais fácil a manutenção desses arquivos. Dessa forma, toda vez que formos alterar alguma configuração de rota não corremos o risco de comprometer alguma configuração essencial do Express.

Vimos também que apesar de salutar, somos obrigados a lembrar de fazermos o require dessas rotas no arquivo de configuração do Express. Podemos esquecer de fazer isso ou até mesmo quebrarmos o arquivo de configuração do Express e foi por isso que usamos o módulo `consign` para fazer o carregamento automático desses módulos sem termos que alterar o arquivo de configuração do Express. Vimos também que é interessante separarmos a lógica da aplicação em um arquivo e a rota que dispara a execução dessa lógica em outra. No final, ficamos com uma estrutura mais fácil de manter.