

NodeJS – Módulo 03

Material de estudo desenvolvido por:
Daniel Teixeira Quadros

Sumário

Aula 01 – Not Found (404).....	2
Aula 02 – Criando um Model.....	4
Aula 03 – Usando o Model (1/2).....	7
Aula 04 – Usando o Model (2/2).....	10
Aula 05 – Usando Mensagens Flash.....	13
Aula 06 – Lidando com erros.....	19
Aula 07 – Listando os Posts.....	20
Aula 08 – Criando o Slug no Post.....	22
Aula 09 – Editar Post.....	24
Aula 10 – Otimizando o Editar.....	29
Aula 11 – Adicionar Tags.....	31
Aula 12 – Editar Tags.....	32
Aula 13 – Visualizar Post Único.....	33
Aula 14 – Inserindo CSS no Template.....	35

Aula 01 – Not Found (404)

Not Found (404) → Página não encontrada.

Para esta função, criaremos um middleware no **app.js**:

```
app.use((req, res, next) => {  
  res.send('Página não encontrada')  
})
```

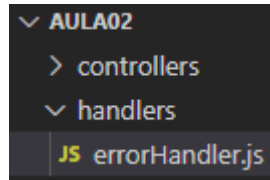
Uma das opções é adicionar este middleware. Se chegar aqui, é porque ele passou do middleware acima que definiu as rotas existentes (até então).

```
app.use(express.json())  
app.use('/', router)  
app.use((req, res, next) => {  
  res.send('Página não encontrada')  
})
```

Obs.: Por padrão, sempre adicionamos os parâmetros 'req', 'res' e 'next' nos middleware, mesmo que a princípio não iremos utilizar todos aqui.

Ainda podemos otimizar este formato, vejamos:

- Criamos uma nova pasta na raiz do nosso projeto chamada de **'handlers'** e dentro dela criamos o arquivo **'errorHandler.js'**:



Neste arquivo adicionaremos códigos que lidam com problemas que estejam relacionados a erros.

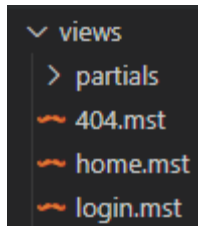
Adicionamos o código a seguir:

```
JS app.js    JS errorHandler.js X    404.mst  
handlers > JS errorHandler.js > notFound > exports.notFound  
1  exports.notFound = (req, res, next) => {  
2    res.status = 404  
3    res.render('404')  
4  }
```

Criamos esta função aqui, e com ela, exportamos a exibição de uma página (404) que iremos criar lá no 'views' e será exibida sempre que uma página não for encontrada.

res.status = 404 → É muito interessante, que além da mensagem que será exibida na tela que está em '404.mst', também seja alterado o status para 404, indicando o tipo de erro (404 = página não encontrada).

Dentro do views criamos a página **404.mst**:



E adicionamos um layout simples:

```
views > 404.mst
1 <h1>Página não Encontrada </h1>
2 <p>Procure em XYZ</p>
```

No app.js:

const errorHandler = require('./handlers/errorHandler') → Importamos o nosso arquivo 'errorHandler' para a nossa aplicação.

Excluímos a função anônima que havíamos criado no app.use que vinha após as rotas e chamamos o nosso **notFound** para a aplicação:

app.use(errorHandler.notFound) → Se passar pelas rotas, o parâmetro 'next' do middleware anterior, trará a execução para esta linha e a variável 'notFound' será executada.

Nosso app.js fica assim:

```
JS app.js > ...
1  const express = require('express')
2  const mustache = require('mustache-express')
3  const router = require('./routes/index')
4  const helpers = require('./helpers')
5  const errorHandler = require('./handlers/errorHandler')
6
7  //Configurações
8  const app = express()
9  app.use((req, res, next)=>{
10     res.locals.h = helpers
11     next()
12 })
13 app.use(express.json())
14 app.use('/', router)
15 app.use(errorHandler.notFound)
16
17 app.engine('mst', mustache(__dirname+'/views/partials','.mst'))
18 app.set('view engine', 'mst')
19 app.set('views', __dirname + '/views')
20
21 module.exports = app
22
```

Ao acessarmos uma rota inexistente da nossa aplicação, a nossa página 404.mst é exibida:

Página não Encontrada

Procure em XYZ

Aula 02 – Criando um Model

Antes de começar a aula 02, o Bonieky avisou que fez as seguintes alteração:

No arquivo **index.js** removeu o **homeController.userMiddleware** da linha 6, deixando o arquivo assim:

```
JS index.js X
routes > JS index.js > ...
1  const express = require('express')
2  const homeController = require('../controllers/homeController')
3  const userController = require('../controllers/userController')
4
5  const router = express.Router()
6  router.get('/', homeController.index)
7  router.get('/users/login', userController.login)
8  router.get('/users/register', userController.register)
9
10 module.exports = router
```

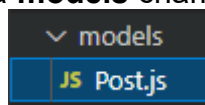
No arquivo **homeController**, foi removido a função anônima da variável **userMiddleware** das linhas 1 até 5, deixando o arquivo assim:

```
JS index.js X JS homeController.js X
controllers > JS homeController.js > index > exports.index
1  exports.index = (req, res)=>{
2    let obj = {
3      pageTitle:"Home",
4      userInfo: req.userInfo
5    }
6    res.render('home', obj)
7  }
```

Lembramos aqui, que para a nossa aplicação, criamos um banco de dados (que ainda está vazio), pensando na estrutura de um blog. E, em todo o blog, obviamente existem posts.

Iremos trabalhar agora com models pensando nesta estrutura. Por isso iremos criar um model que trabalhe com os posts, para quando criar, excluir ou alterar estes posts utilizaremos este model.

Para isto, criaremos um arquivo dentro da pasta **models** chamado de **Post.js**:



Obs.: Para o model, geralmente se utiliza a primeira letra em maiúscula.

const mongoose = require('mongoose') → Aqui, criamos a variável 'mongoose' que utilizaremos para nos conectarmos com o mongo DB utilizando os recursos do mongoose.

mongoose.Promise = global.Promise → Função atualizada que todo o navegador tem e o Node também tem, e estamos passando para o Mongoose a versão mais atualizada da forma de comunicação que podemos utilizar.

```
const mongoose = require('mongoose')
mongoose.Promise = global.Promise
```

MUITO IMPORTANTE ENTENDER:

O Mongo DB tem a fama de ser um Banco de Dados mais flexível. Recomendação → Aqui, nos criaremos uma estrutura pré-definida para termos essa estrutura de uma forma mais organizada. Para isto utilizaremos um **schema**. Criaremos uma variável chamada postSchema para isto.

Precisamos pensar, na estrutura de linhas e colunas que esse nosso banco de dados precisa ter, pensando aqui, mais especificamente nos posts:

Colunas:

- Título
- Corpo
- Tags
- Slug

Criação da estrutura pré-definida (Schema):

```
const postSchema = new mongoose.Schema({
  title:{
    type:String,
    trim:true,
    required:'O post precisa de um título'
  },
  slug:String,
  body:{
    type:String,
    trim:true
  },
  tags:[String]
})
```

const postSchema = new mongoose.Schema({

title:{ → Aqui, podemos criar um objeto para definirmos o título de forma mais específica. porém, se quisermos, podemos também apenas criar assim (title:String)

type:String, → O tipo

trim:true, → Tira espaços excedentes do que for digitado pelo usuário para que no título, o usuário não encha de espaços em branco no meio do texto do título

required:'O post precisa de um título' → Obriga o usuário, durante a criação de um post, a preencher o título. Aqui, adicionamos uma mensagem neste item, porém, se não quisermos adicionar uma mensagem, poderíamos ter preenchido o require apenas com a informação 'true', desta forma: **require:true**.

slug:String, → Endereço do post específico.

body:{ → Corpo do post.

type:String, → Tipo.

trim:true → Remove os espaços excedentes.

tags:[String] → Para as tags, que geralmente (não obrigatoriamente), que se adicionam várias, criamos um array e definimos o tipo de conteúdo que será armazenado aqui. 'String'.

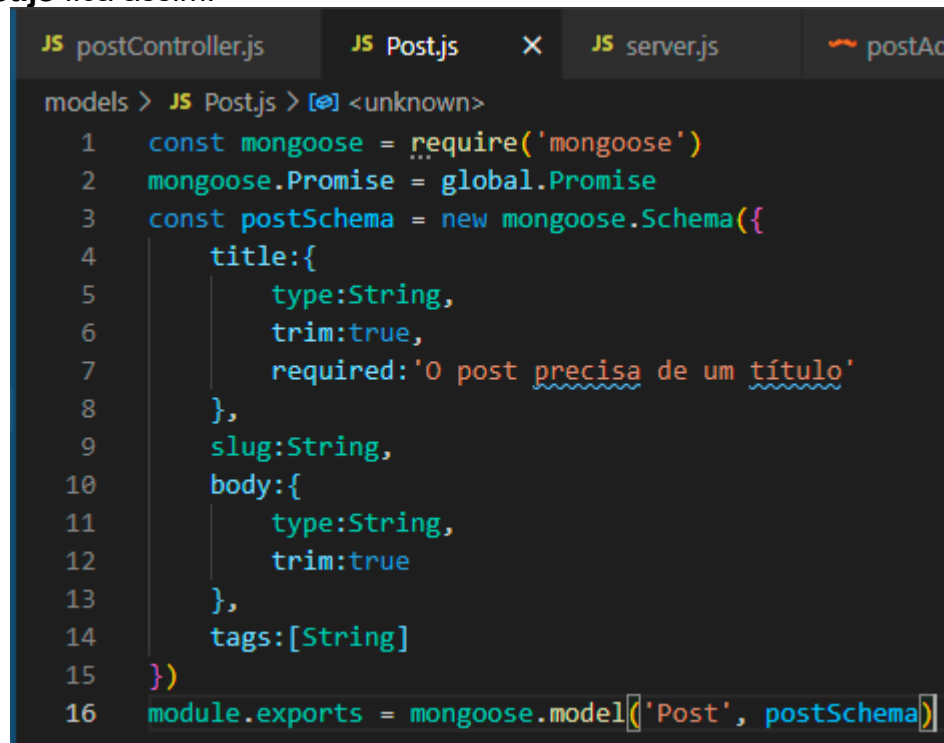
Até aqui, apenas configuramos o básico do nosso model. Abaixo iremos criá-lo.

module.exports = mongoose.model('Post', postSchema) → Comando para criar o nosso módulo. Utilizamos o mongoose.model e adicionamos dois parâmetros:

Post → Nome do model

postSchema → Adicionamos a estrutura que criamos acima. Lembrando que no Mongo DB, esta estrutura pré-definida é opcional.

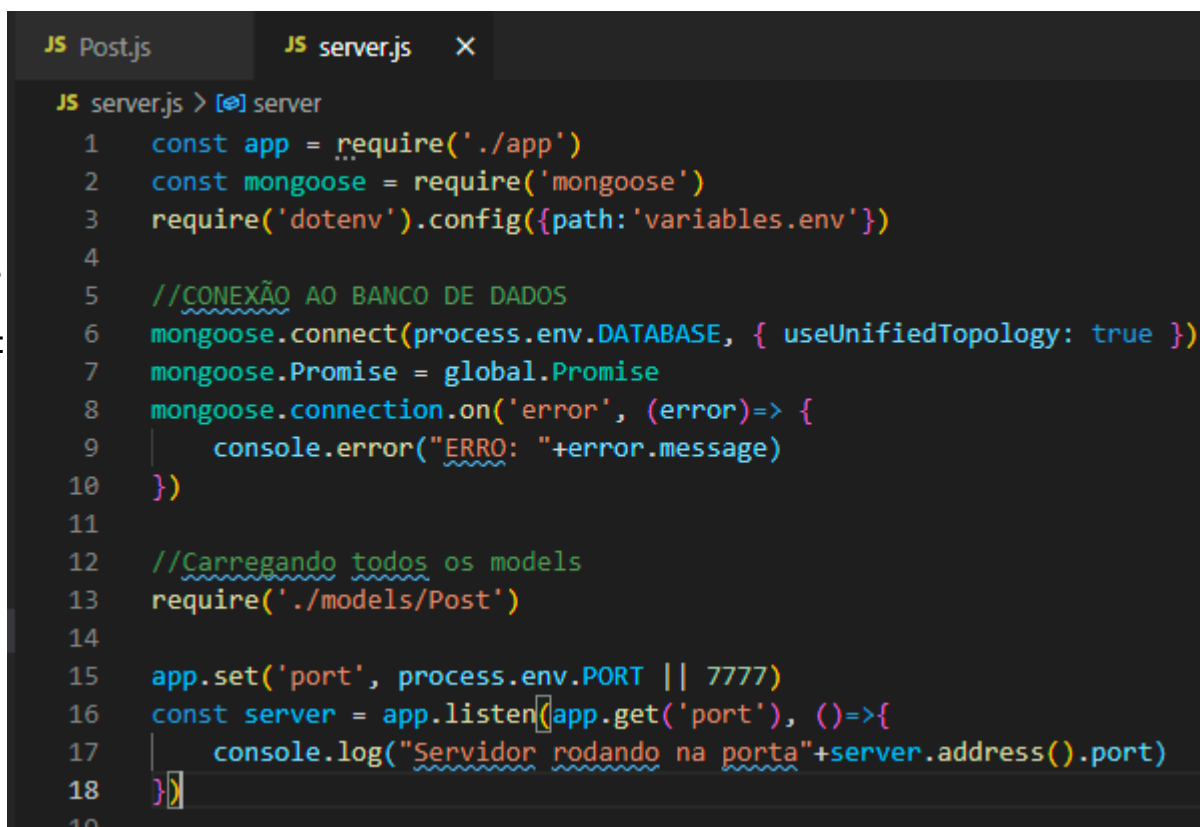
Até aqui, o nosso **Post.js** fica assim:



```
JS postController.js JS Post.js X JS server.js postAd
models > JS Post.js > [?] <unknown>
1  const mongoose = require('mongoose')
2  mongoose.Promise = global.Promise
3  const postSchema = new mongoose.Schema({
4    title:{
5      type:String,
6      trim:true,
7      required:'O post precisa de um título'
8    },
9    slug:String,
10   body:{
11     type:String,
12     trim:true
13   },
14   tags:[String]
15 })
16 module.exports = mongoose.model('Post', postSchema)
```

Agora, precisamos carregar este model no nosso servidor. Para isto adicionamos um **require** no **server.js**:
`require('./models/Post')`

Deixando o nosso **server.js** assim:



```
JS Post.js JS server.js X
JS server.js > [?] server
1  const app = require('./app')
2  const mongoose = require('mongoose')
3  require('dotenv').config({path:'variables.env'})
4
5  //CONEXÃO AO BANCO DE DADOS
6  mongoose.connect(process.env.DATABASE, { useUnifiedTopology: true })
7  mongoose.Promise = global.Promise
8  mongoose.connection.on('error', (error)=> {
9    console.error("ERRO: "+error.message)
10  })
11
12  //Carregando todos os models
13  require('./models/Post')
14
15  app.set('port', process.env.PORT || 7777)
16  const server = app.listen(app.get('port'), ()=>{
17    console.log("Servidor rodando na porta"+server.address().port)
18  })
```

Aula 03 – Usando o Model (1/2)

Primeiramente aqui, criaremos uma página para adicionar os nossos posts, para isto, acessaremos o nosso arquivo **helpers.js** e excluiríamos as linhas 4, 5 e 6 que são de rotas que não estão sendo utilizadas, deixando o nosso helpers inicialmente assim:

```
JS helpers.js > [🔗] menu
1  exports.defaultPageTitle = "Site ABC"
2  exports.menu = [
3    {name:'Home', slug:'/'},
4    {name:'Login',slug:'/users/login'}
5  ]
```

Aqui, criamos a seguinte função:

{name:'Adicionar Post',slug:'/post/add'} → Criamos aqui a nossa função para criar o link para ser adicionado novo post.

A partir daqui, precisamos criar a nossa **rota** 'post/add' e criarmos um **controller** de post

Nosso helper.js fica assim:

```
JS helpers.js > [🔗] menu
1  exports.defaultPageTitle = "Site ABC"
2  exports.menu = [
3    {name:'Home', slug:'/'},
4    {name:'Login',slug:'/users/login'},
5    {name:'Adicionar Post',slug:'/post/add'}
6  ]
```

Vamos para o nosso **index.js** para fazermos a importação (require) e criarmos a rota para o **postController** que iremos criar posteriormente:

Para isto, adicionamos as seguintes linhas no index.js:

const postController = require('../controllers/postController') → Aqui, estamos criando o **'require'** deste controller que ainda não existe, mas criaremos ele logo a seguir.

router.get('/post/add', postController.add) → Aqui criamos a nossa rota para a adição de posts e apontamos para um **postController** que iremos criar com um método que chamaremos de **add**

nosso index.js fica assim:

```
routes > JS index.js > [0] postController
1  const express = require('express')
2  const homeController = require('../controllers/homeController')
3  const userController = require('../controllers/userController')
4  const postController = require('../controllers/postController')
5
6  const router = express.Router()
7  router.get('/', homeController.index)
8  router.get('/users/login', userController.login)
9  router.get('/users/register', userController.register)
10 router.get('/post/add', postController.add)
11
12 module.exports = router
```

Agora, é o momento em que criaremos o arquivo **postController.js** dentro da pasta **controllers**:

```
▼ controllers
JS homeController.js
JS postController.js
JS userController.js
```

postController:

exports.add = (req, res) => { → Aqui, efetivamente, iremos exportar o nosso postController
res.render('postAdd') → Renderizamos aqui a página para adicionar o post que iremos criar no views

postController.js:

```
controllers > JS postController.js > add > exports.add
1  exports.add = (req, res) => {
2    res.render('postAdd')
```

Agora, no views, criaremos a página para adicionar os posts. Criaremos o arquivo **postAdd.mst** dentro do **views**:

```
views > postAdd.mst
1  {{> header}}
2
3  <h2>Adicionar Post</h2>
4  <form method="POST">
5    <label>
6      Título:
7      <input type="text" name="title">
8    </label>
9    <br>
10   <label>
11     Corpo:
12     <textarea name="body"></textarea>
13   </label>
14   <br>
15   <label>
16     Tags:
17   </label>
18   <br>
19   <input type="submit" value="Salvar">
20 </form>
```

E criamos a seguinte estrutura HTML:

Aqui, a nossa página já está criada, porém, criamos o nosso formulário para ser enviado para o servidor via **POST** e não via **GET**. Para que este envio funcione, precisaremos editar mais uma vez o nosso index.js para adicionarmos essa rota:

No index.js:

router.post('/post/add', postController.addAction) → Adicionando a rota utilizando o método post para adicionar o post criado no banco de dados.

Obs.: existe uma convenção que é utilizada para a nomenclatura que segue a seguinte estrutura. nome da tela (**add**) + a palavra **Action**. O add é a tela, e o Action é o recebimento dos dados daquela tela (**addAction**)

Agora precisamos ir no nosso postController.js para criarmos o export do addAction

O index.js fica assim:

```
routes > JS index.js > ...
1  const express = require('express')
2  const homeController = require('../controllers/homeController')
3  const userController = require('../controllers/userController')
4  const postController = require('../controllers/postController')
5
6  const router = express.Router()
7  router.get('/', homeController.index)
8  router.get('/users/login', userController.login)
9  router.get('/users/register', userController.register)
10 router.get('/post/add', postController.add)
11 router.post('/post/add', postController.addAction)
12
13 module.exports = router
```

No postController.js adicionamos o seguinte comando:

```
exports.addAction = (req, res) => {
  req.body
}
```

Lembrando que aqui estamos utilizando o req.body porque se trata de uma requisição POST, se fosse uma requisição GET, utilizaríamos o req.query

Após a explicação do req.body acima o Bonieky apagou este comando e terminamos esta aula com o postController.js assim:

```
controllers > JS postController.js > addAction > exports.addAction
1  exports.add = (req, res) => {
2    |    res.render('postAdd')
3  }
4  exports.addAction = (req, res) => {}
5  |
6  }
```

Aula 04 – Usando o Model (2/2)

Continuando no nosso postController adicionamos os seguintes comandos:

const mongoose = require('mongoose') → Para o nosso addAction funcionar, precisamos nos conectar com o banco de dados, por isto criamos a variável mongoose aqui e importamos os recursos do mongoose para ela.

const Post = mongoose.model('Post') → Aqui, utilizamos o mongoose para instanciar o nosso Post

res.json(req.body) → Utilizamos esta linha apenas para exibir os dados que estamos recebendo no addAction

postController.js:

```
controllers > JS postController.js > addAction > exports.addAction
1  const mongoose = require('mongoose')
2  const Post = mongoose.model('Post')
3
4  exports.add = (req, res) => {
5    |    res.render('postAdd')
6  }
7  exports.addAction = (req, res) => {
8    |    res.json(req.body)
9  }
```

Pela fato de o javascript ser uma linguagem síncrona, tivemos um erro aqui, para corrigir este erro, precisamos editar o nosso arquivo **server.js**:

Removemos o conteúdo da **linha 1 (const app = require('./app'))** e **colocamos** este conteúdo **após a requisição do Post, na linha 13.**

Desta forma, apenas depois de puxarmos(rodarmos) os nossos model é que iremos rodar o nosso aplicativo.

E o server.js fica assim:

```

JS server.js > [?] server
1  const mongoose = require('mongoose')
2  require('dotenv').config({path: 'variables.env'})
3
4  //CONEXÃO AO BANCO DE DADOS
5  mongoose.connect(process.env.DATABASE, { useUnifiedTopology: true })
6  mongoose.Promise = global.Promise
7  mongoose.connection.on('error', (error)=> {
8    console.error("ERRO: "+error.message)
9  })
10
11  //Carregando todos os models
12  require('./models/Post')
13  const app = require('./app')
14
15  app.set('port', process.env.PORT || 7777)
16  const server = app.listen(app.get('port'), ()=>{
17    console.log("Servidor rodando na porta"+server.address().port)
18  })

```

Agora, no app.js precisamos adicionar um comando para que o envio dos dados para o banco tenha o formato necessário para ser recebido:

app.use(express.urlencoded({extended:true})) → Aqui ativamos o urlencoded para que as informações sejam efetivamente montadas, padronizadas no formato json para que possam ser recebidas no banco de dados. **Linha 14**

```

JS app.js > [?] <unknown>
1  const express = require('express')
2  const mustache = require('mustache-express')
3  const router = require('./routes/index')
4  const helpers = require('./helpers')
5  const errorHandler = require('./handlers/errorHandler')
6
7  //Configurações
8  const app = express()
9  app.use((req, res, next)=>{
10    res.locals.h = helpers
11    next()
12  })
13  app.use(express.json())
14  app.use(express.urlencoded({extended:true}))
15  app.use
16  app.use
17
18  app.engine('html', require('mustache').render)
19  app.set('view engine', 'html')
20  app.set('views', path.resolve(__dirname, '../views'))
21
22  module.

```

Cabeçalho

- [Home](#)
- [Login](#)
- [Adicionar Post](#)

No navegador, já podemos visualizar o formato de envio dos dados do post:

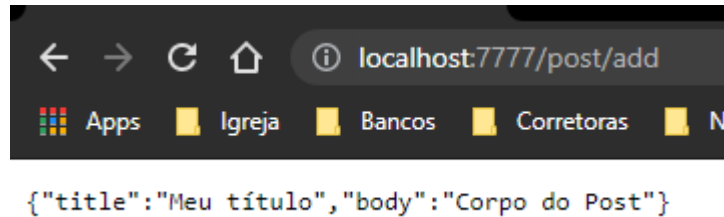
Adicionar Post

Título:

Corpo:

Tags:

Após clicar em Salvar:



Voltamos ao `postController` e realizamos as seguintes modificações:

Excluimos o `res.json(req.body)` da linha 8 que utilizamos apenas para teste, e no nosso `exports.addAction` adicionamos os seguintes comandos que nos permitirão usar o nosso `model` para salvarmos efetivamente os nossos dados no banco de dados:

`const post = new Post (req.body)` → Aqui criamos um novo `post`. Podemos criar um objeto e passarmos o que quisermos. Porém, como os itens do nosso objeto **`postSchema (Post.js)`** já tem os mesmos nomes que utilizamos nos `'name'` dos itens HTML do arquivo **`postAdd.mst`**, podemos enviar diretamente a requisição via POST desta forma (`req.body`)

`post.save()` → Este comando que irá realmente salvar o POST. e faz a comunicação com o Mongo DB.

Depois de salvo, podemos, se quisermos, redirecionar para a página inicial. por exemplo.

`res.redirect('/')` → Este comando redireciona para a página inicial.

Até aqui tudo bem, porém tem um pequeno detalhe que é o seguinte:

Como já vimos anteriormente, o javascript é síncrono. Portanto ele dará o `'save'` e ele vai fazer a comunicação com o Mongo DB, que é uma estrutura externa ao javascript. Isso significa que teremos que esperar uma resposta do Mongo DB para depois continuarmos a nossa aplicação.

Por isto, neste momento é importante entender que o `'save'` retorna uma **Promise** (Promessa) → Conteúdo já visto no curso de javascript.

Por isso aqui utilizaremos o conjunto de comandos **`async await`**

```
exports.addAction = async (req, res) => {  
  const post = new Post (req.body)  
  await post.save()  
  res.redirect('/')  
}
```

`async await` → Desta forma, ele vai dar o `'save'` vai esperar a resposta do `save` e só então ele irá redirecionar para a página inicial.

Nosso postController fica assim:

```
controllers > JS postController.js > addAction > exports.addAction
1  const mongoose = require('mongoose')
2  const Post = mongoose.model('Post')
3
4  exports.add = (req, res) => {
5    res.render('postAdd')
6  }
7  exports.addAction = async (req, res) => {
8    const post = new Post (req.body)
9    await post.save()
10   res.redirect('/')
11 }
```

Aula 05 – Usando Mensagens Flash

Mensagens Flash são mensagens que configuramos para que o sistema retorne para o usuário alguma informação que é importante para ele porém, normalmente o sistema não mostra.

Por exemplo: Quando um usuário criar um novo post no nosso sistema, poderíamos mostrar alguma mensagem pra ele informando que o post foi salvo, ou foi criado com sucesso.

Vejamos o exemplo na prática:

Para podermos criar as mensagens flash iremos instalar três bibliotecas distintas. Para isto, acessaremos o terminal.

As bibliotecas que iremos instalar são as seguintes:

- cookie-parser
- express-session
- express-flash

Para isto abrimos um novo terminal e executamos o seguinte comando para instalarmos as três ao mesmo tempo:

npm install cookie-parser express-session express-flash --save

Temos o seguinte resultado:

```
PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02> npm install cookie-parser express-session express-flash --save
npm WARN aula02@1.0.0 No description
npm WARN aula02@1.0.0 No repository field.
npm WARN aula02@1.0.0 No license field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ express-flash@0.0.2
+ cookie-parser@1.4.5
+ express-session@1.17.2
added 10 packages from 8 contributors and audited 213 packages in 5.309s

13 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02>
```

Agora, com as bibliotecas instaladas vamos importá-las para a nossa aplicação no **app.js**

OBS IMPORTANTÍSSIMA: AO ABRIR O app.js COM O BONIEKY, PERCEBI ALGUMAS ALTERAÇÕES NESTE AQUIVO QUE NÃO HAVIA SIDO COMENTADAS OU EU NÃO HAVIA PERCEBIDO. SÃO ELAS:

Adição do comando `const mongoose = require('mongoose')` na linha 2.

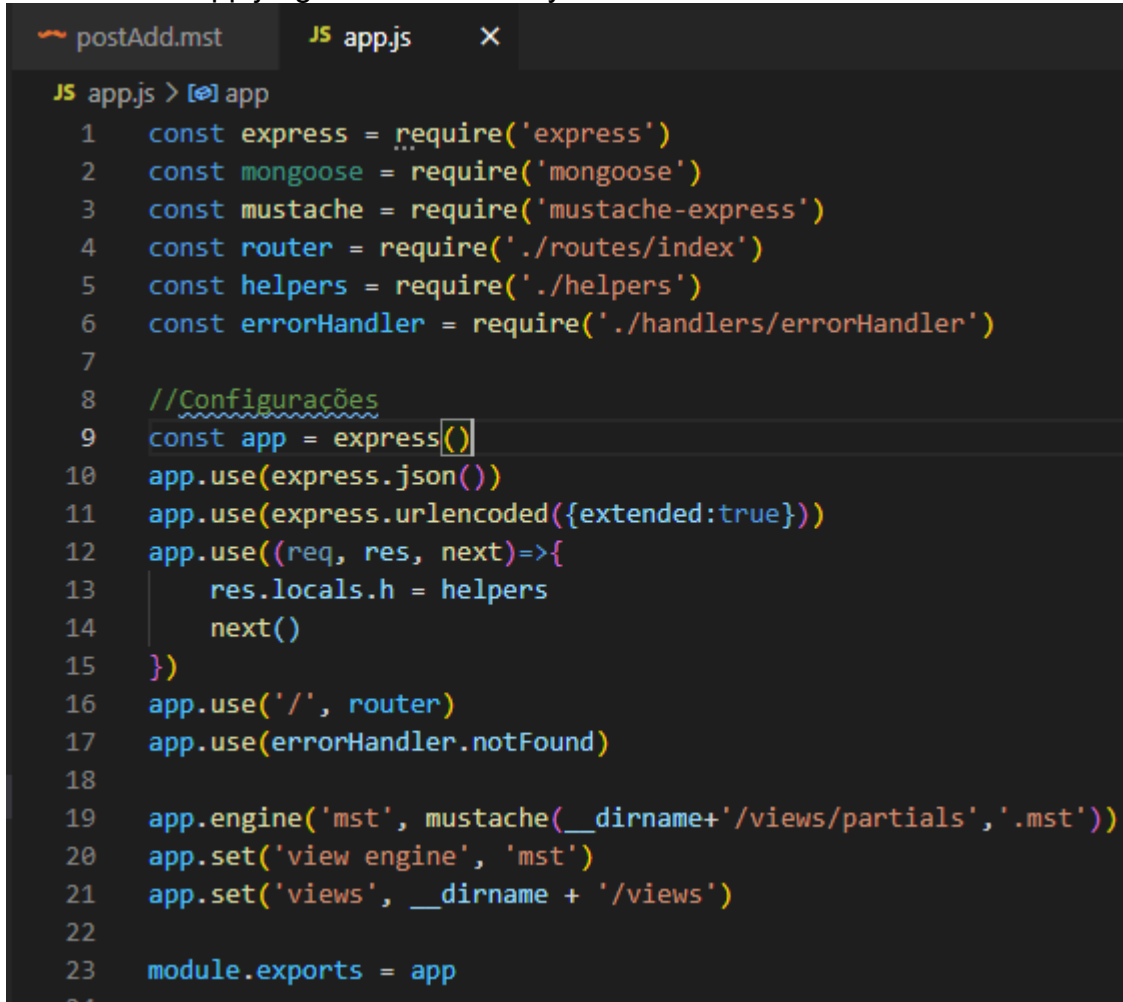
Deslocamento das linhas 14 e 15 para a 10 e 11

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

`app.use(express.urlencoded({extended:true}))`

Colocadas após a linha `const app = express()`

Partimos daqui com o nosso app.js igual ao do bonieky conforme abaixo:



```
JS app.js > [X] app
1  const express = require('express')
2  const mongoose = require('mongoose')
3  const mustache = require('mustache-express')
4  const router = require('./routes/index')
5  const helpers = require('./helpers')
6  const errorHandler = require('./handlers/errorHandler')
7
8  //Configurações
9  const app = express()
10 app.use(express.json())
11 app.use(express.urlencoded({extended:true}))
12 app.use((req, res, next)=>{
13     res.locals.h = helpers
14     next()
15 })
16 app.use('/', router)
17 app.use(errorHandler.notFound)
18
19 app.engine('mst', mustache(__dirname+'/views/partials','.mst'))
20 app.set('view engine', 'mst')
21 app.set('views', __dirname + '/views')
22
23 module.exports = app
```

Iremos agora importar essas bibliotecas que instalamos para a nossa aplicação (app.js)

Para isto adicionaremos as seguintes variáveis:

`const cookieParse = require('cookie-parse')`

`const session = require('express-session')`

`const flash = require('express-flash')`

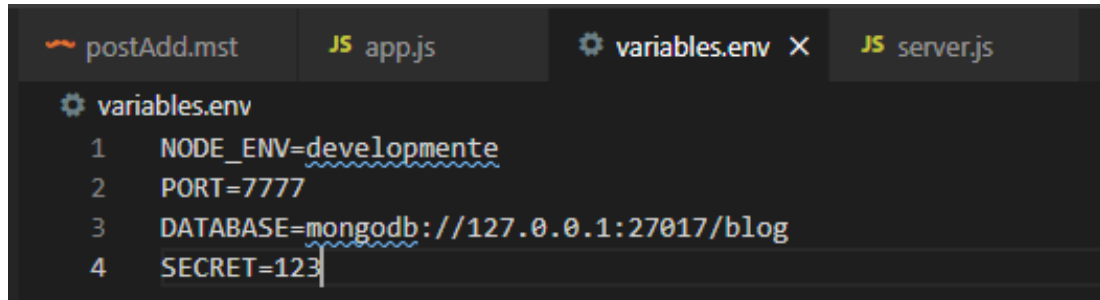
Agora, para utilizarmos essas três bibliotecas, logo abaixo dos códigos que mexem com as requisições vamos fazer as primeiras configurações.

Antes de habilitarmos o Flash que é a funcionalidade que queremos utilizar, precisamos habilitar o cookie e o session.

app.use(cookieParser(process.env.SECRET)) → Comando que habilita o cookie. Iremos passar aqui uma chave secreta (Passar esta chave é opcional). Ele usa esta chave para decodificar o nosso cookie. Como é uma chave secreta específica, o ideal é que coloquemos ela nas nossas variáveis de ambiente (variables.env)

```
app.use(cookieParser(process.env.SECRET))
```

O SECRET que iremos criar para o exercício, terá a chave 123:



```
variables.env
1  NODE_ENV=desenvolvimento
2  PORT=7777
3  DATABASE=mongodb://127.0.0.1:27017/blog
4  SECRET=123
```

app.use(session({ → Habilitando a session → Aqui, passaremos um objeto com algumas configurações específicas que precisamos passar. A primeira delas é o SECRET também:

secret:process.env.SECRET,

E mais duas opções:

resave:false, → Vai dizer para a sessão, caso não tenha ocorrido modificação nesta sessão, que ela não precisa ser destruída e recriada. Se não colocarmos esta sessão, por padrão vem configurada como true.

Isto é necessário porque cada vez que eu entro em uma página, mesmo que não aconteça nenhuma alteração, essa sessão será re-salvada, ou seja, reescrita. Inclusive esta orientação de utilizar resave:false está na documentação desta biblioteca.

saveUninitialized:false → Também desabilitamos esta função para que quando iniciamos uma função mas não salvamos absolutamente nada, não faz sentido salvarmos ela se ela não possui nenhum dado para ser salvo.

```
app.use(session({
  secret:process.env.SECRET,
  resave:false,
  saveUninitialized:false
}))
```

Habilitando o Flash:

```
app.use(flash())
```

app.use(flash()) → Aqui habilitamos o Flash e já podemos utilizá-lo.

É importante que deixemos as mensagens do Flash a nível global, ou seja, que possamos acessá-las em qualquer página da nossa aplicação. Para isto, iremos adicionar mais uma opção no locals.

res.locals.flashes = req.flash() → Esta linha permite que o nosso Flash tenha aplicação global e resolvemos aqui, chamar esta habilidade de flashes. Fazemos uma requisição do Flash req.flash().

```
app.use(flash())
app.use((req, res, next)=>{
  res.locals.h = helpers
  res.locals.flashes = req.flash()
  next()
})
```

No postController.js faremos algumas alterações:

Agora, aqui no postController, após salvar um novo post, podemos utilizar uma flash mensagem.

req.flash('success','Post salvo com sucesso') → Este comando tem dois parâmetro: O primeiro é o tipo de mensagem. (Normalmente se usam três tipos: - success, - error, - info) O segundo parâmetro é a mensagem que queremos enviar.

```
exports.addAction = async (req, res) => {
  const post = new Post (req.body)
  await post.save()
  req.flash('success', 'Post salvo com sucesso')
  res.redirect('/')
}
```

Agora, que estamos com as configurações prontas, precisamos criar a possibilidade de exibi-la.

Um local interessante para isto seria no nosso cabeçalho, já que preparamos ele para ser exibido em quase todas as páginas da nossa aplicação.

header.mst →

```
{{#flashes}}
  {{#success}}
    <div>{{.}}</div>
  {{/success}}
{{/flashes}}
```

{{#flashes}} → Estamos verificando se há mensagem para ser exibida.

Dentro dos flashes terão os tipos de mensagens que nós salvamos. Neste caso, salvamos apenas uma do tipo success.

Também podemos salvar diversas mensagens, pois elas vêm em forma de array.

{{#success}} → Criamos um loop em success.

<div>{{.}}</div> → Este ponto dentro das chaves faz com que o loop vá exibindo todas as mensagens que forem recebidas neste array aqui em success

Agora, ao adicionarmos um post com sucesso, a mensagem flash que criamos é exibida no navegador:

Cabeçalho

- [Home](#)
- [Login](#)
- [Adicionar Post](#)

Post salvo com sucesso

Página Home

No final da aula, preparamos o nosso header.mst para exibir as mensagens do tipo error e info, embora não tenhamos criado elas ainda

header.mst:

```
JS postController.js  header.mst X  JS app.js
views > partials > header.mst
1  <html>
2  <head>
3    <title>{{pageTitle}} {{h.defaultPageTitle}}</title>
4  </head>
5  <body>
6    <h1>Cabeçalho</h1>
7    <hr>
8    <ul>
9      {{#h.menu}}
10     <li><a href="{{slug}}">{{name}}</a></li>
11   {{/h.menu}}
12 </ul>
13 <hr>
14 {{#flashes}}
15   {{#success}}
16     <div>{{.}}</div>
17   {{/success}}
18   {{#error}}
19     <div>{{.}}</div>
20   {{/error}}
21   {{#info}}
22     <div>{{.}}</div>
23   {{/info}}
24 {{/flashes}}
25 </body>
```

app.js:

```
JS postController.js  header.mst  JS app.js  X

JS app.js > ...
1  const express = require('express')
2  const mongoose = require('mongoose')
3  const mustache = require('mustache-express')
4  const cookieParser = require('cookie-parser')
5  const session = require('express-session')
6  const flash = require('express-flash')
7  const router = require('./routes/index')
8  const helpers = require('./helpers')
9  const errorHandler = require('./handlers/errorHandler')
10 //Configurações
11 const app = express()
12 app.use(express.json())
13 app.use(express.urlencoded({extended:true}))
14 app.use(cookieParser(process.env.SECRET))
15 app.use(session({
16   secret:process.env.SECRET,
17   resave:false,
18   saveUninitialized:false
19 }))
20 app.use(flash())
21 .../.../...
22 JS postController.js  X  header.mst  JS app.js
23 controllers > JS postController.js > addAction > exports.addAction
24 1  const mongoose = require('mongoose')
25 2  const Post = mongoose.model('Post')
26 3
27 4  exports.add = (req, res) => {
28 5    res.render('postAdd' mst'))
29 6  }
30 7  exports.addAction = async (req, res) => {
31 8    const post = new Post (req.body)
32 9    await post.save()
10 10   req.flash('success','Post salvo com sucesso')
11 11   res.redirect('/')
12 12 }
```

postController.js

Aula 06 – Lidando com erros

Nesta aula vamos analisar alguns erros que podem vir a acontecer nesta aplicação.

O Primeiro erro simulado foi o de tentar salvar um novo post sem nenhuma informação preenchida (Título, Corpo). Neste teste, o navegador ficou tentando carregar, porém, como os dados estavam vazios, o navegador não recebeu retorno do Mongo DB.

Para evitarmos este tipo de erro, vamos trabalhar no nosso **postController.js**

`await post.save()` → O erro que temos que evitar que aconteça é nesta linha aqui, pois é ela que fica esperando que a informação seja salva no Mongo DB para só depois dar continuidade na aplicação.

Para isto, adicionaremos os seguintes comando no postController.js:

```
try{
  await post.save()
} catch(error){
  req.flash('error', 'Erro: '+error.message)
  return res.redirect('/post/add')
}
```

try → Este comando informa que ao invés de haver a obrigatoriedade de ocorrer o salvamento de um novo post no Mongo DB, haverá a tentativa de salvamento.

catch(error) → O catch pega eventuais erros que possam ocorrer na tentativa de salvamento de um novo post.

Obs.: O catch só deverá ser executado se o que estiver dentro do try (`await post.save()`) não for executado.

error → Se ocorrer algum erro, a primeira coisa que iremos fazer aqui é informar o usuário que ocorreu um erro

req.flash('error', 'Erro: '+error.message) → O req.flash é utilizado aqui novamente para trazer uma mensagem flash que neste caso utilizamos o tipo **'error'** (primeiro parâmetro). No segundo parâmetro, escrevemos a mensagem que queremos que seja apresentada para o usuário, que neste caso, escrevemos para ele o seguinte:

'Erro: '+error.message → O texto 'Erro: ' mais (concatenação) +error.message → Uma mensagem de erro que será enviada pelo próprio sistema(javascript)

return res.redirect('post/add') → Após enviarmos a mensagem para o usuário, precisamos dar seguimento na aplicação para que o erro pare de ocorrer. Por isto, redirecionamos aqui o usuário para a mesma página, a de adicionar post.

Obs.: Utilizamos o **return** aqui, para que o nosso addAction não siga para as duas próximas linhas (imprimir a mensagem de success e redirecionar para a página inicial).

Se quisermos, podemos substituir o segundo parâmetro do req.flash apenas por um texto:

```
req.flash('error', 'Ocorreu um erro! Tente novamente mais tarde!')
```

Concluimos a aula com o postController assim:

```
1  const mongoose = require('mongoose')
2  const Post = mongoose.model('Post')
3
4  exports.add = (req, res) => {
5    res.render('postAdd')
6  }
7  exports.addAction = async (req, res) => {
8    const post = new Post (req.body)
9    try{
10     await post.save()
11   } catch(error){
12     req.flash('error', 'Ocorreu um erro! Tente novamente mais tarde!')
13     return res.redirect('/post/add')
14   }
15   req.flash('success','Post salvo com sucesso')
16   res.redirect('/')
17 }
```

Aula 07 – Listando os Posts

Na aula, listaremos os posts na página home.

Para listarmos os posts iremos precisar do mongoose e do nosso model

Começamos editando o homeController.js e adicionamos as seguintes linhas no início do arquivo:

```
const mongoose = require('mongoose')//importamos o mongoose
const Post = mongoose.model('Post')//Instanciamos o model
```

Como iremos listar os posts na página home, faremos isto no item index. E como vamos acessar o banco de dados, utilizaremos a função **async await**.

Para prosseguir, removemos a linha `userInfo: req.userInfo` do nosso objeto **obj**.

let responseJson = { → O Bonieky orientou que trocássemos o nome desta variável '**obj**' por '**responseJson**' apenas por questões de boas práticas de nomeação de variável. Com esta troca de nome aqui, trocamos também o nome dela na linha **res.render**.

HomeController:

```
const mongoose = require('mongoose')
const Post = mongoose.model('Post')

exports.index = async (req, res) => {
  let responseJson = {
    pageTitle: "Home",
    posts: []
  }
  const posts = await Post.find()
  responseJson.posts = posts
  res.render('home', responseJson)
}
```

Posts:[] → Criamos este item aqui, como um array vazio, para receber os posts do banco de dados

const posts = await Post.find() → Criamos a variável posts e adicionamos a ela o Post(Nosso model).find. O .find busca todos os itens post do banco de dados. Como estamos lhe dando com o banco de dados, é aqui que inserimos o await.

responseJson.posts = posts → Agora que já pegamos todos os posts com o .find e colocamos na variável posts, adicionamos estes posts em responseJson.posts

Agora, para podermos visualizar os posts, vamos editar o home.mst adicionando as seguintes linhas para criarmos um contador e listarmos todos os posts existentes:

Cabeçalho

- [Home](#)
- [Login](#)
- [Adicionar Post](#)

Encerramos com o home.mst assim:

Página Home

Meu título

Corpo do post

E no navegador temos o seguinte resultado com todos os posts na nossa home:

Segundo Post

Assunto qualquer

Aula 08 – Criando o Slug no Post

Slug é o endereço de um post específico. No objeto postSchema que criamos no arquivo Post.js, já foi criado o item slug, porém não utilizamos ele ainda.

Para trabalharmos com o slug, iremos instalar mais uma dependência no Node. Para isto, acessamos o terminal e executamos o seguinte comando:

npm install slug --save

```
PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02> npm install slug --save
npm WARN aula02@1.0.0 No description
npm WARN aula02@1.0.0 No repository field.
npm WARN aula02@1.0.0 No license field.
npm WARN optional SKIPPING OPTIONAL DEPENDENCY: fsevents@2.3.2 (node_modules\fsevents):
npm WARN notsup SKIPPING OPTIONAL DEPENDENCY: Unsupported platform for fsevents@2.3.2: wanted {"os":"darwin","arch":"any"} (current: {"os":"win32","arch":"x64"})

+ slug@5.0.1
added 1 package from 1 contributor and audited 214 packages in 3.79s

13 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02> |
```

No arquivo **Post.js** faremos as seguintes modificações:

const slug = require('slug') → Criamos a variável **slug** e importamos os recursos da dependência slug para esta variável.

Após a variável postSchema criamos a seguinte função:

```
postSchema.pre('save', function(next){
  if(this.isModified('title')){
    this.slug = slug(this.title, {lower:true})
  }
  next()
})
```

postSchema.pre('save', function(next){

.pre → Função que permite fazer alteração nos dados antes que alguma outra coisa aconteça. **Esta função te dois parâmetros.**

1º Parâmetro → Informamos a condição, ou seja, antes de.

Save → Antes de salvar

2º Parâmetro → É o que queremos que ela execute.

function → Não criaremos uma função anônima, porque aqui, precisaremos acessar o **'this'** (this é um elemento do próprio objeto.), quando eu estiver salvando um post, por exemplo, eu quero acessar as informações daquele post específico.

if(this.isModified('title')){ → Este **'if'** juntamente com a função **'isModified'** verifica se houve alguma modificação no título, para os casos que o post for modificado mas o seu título não for, não seja executado a alteração de título.

Obs.: Na criação de um **post novo**, ele verifica que o 'title' estava vazio e recebeu algum texto, neste caso, ele considera que foi realizada alguma modificação e altera o slug.

this.slug = slug(this.title, {lower:true}) → Alteramos o slug do item que estamos prestes a salvar colocando o título recebido do usuário como slug. O parâmetro {lower:true} transforma todas as letras em minúsculas, para criar um padrão de slug.

next → Único parâmetro da função, o next vai dizer para a função que após ela ser executada, ela pode seguir para o próximo passo, neste caso, salvar.

Para prosseguirmos, teremos que excluir todos os posts que temos salvo no nosso banco de dados, pois eles não possuem slug.

Para vermos a funcionalidade do slug, criamos novos posts e adicionamos o seguinte comando no home.mst:

{{title}} → Aqui, criamos um link com o título utilizando o slug como link do post.

```

{{>header}}
<h1>Página Home</h1>
{{#posts}}
  <h3><a href="/post/{{slug}}">{{title}}</a></h3>
  <p>{{body}}</p>
  <hr>
{{/posts}}
```

No Navegador, agora, visualizamos os posts assim:

Cabeçalho

- [Home](#)
- [Login](#)
- [Adicionar Post](#)

Página Home

[Título do Post 1](#)

Corpo do Post 1

[Título do Post 2](#)

Corpo do post 2.

E ao clicarmos no link do título, vemos o resultado do slug na URL:

```
localhost:7777/post/titulo-do-post-1
```

Aula 09 – Editar Post

Primeiramente, para editarmos um post, vamos criar um botão para permitir que o usuário edite. Para isto vamos editar o home.mst:

Adicionamos a seguinte linha:

`[Editar]` → Botão para edição de um post específico.

```
{{>header}}
<h1>Página Home</h1>
{{#posts}}
  <h3><a href="/post/{{slug}}">{{title}}</a></h3>
  <p>{{body}}</p>
  <a href="/post/{{slug}}/edit">[ Editar ]</a>
  <hr>
{{/posts}}
```

Criado o botão editar, precisamos agora, criar a rota e a página de edição para que esta função aconteça.

Para isto, vamos para o index.js para criar esta rota:

`router.get('/post/:slug/edit', postController.edit)` → Aqui estamos criando a rota para a edição de cada título específico e o parâmetro `'slug'` torna este trecho do caminho dinâmico, para que não haja a necessidade de criar uma rota para cada post.

Obs.: Direcionamos esta rota para o postController para uma função que criaremos com o nome de **edit**.

```
routes > JS index.js > [❌] <unknown>
1  const express = require('express')
2  const homeController = require('../controllers/homeController')
3  const userController = require('../controllers/userController')
4  const postController = require('../controllers/postController')
5
6  const router = express.Router()
7  router.get('/', homeController.index)
8  router.get('/users/login', userController.login)
9  router.get('/users/register', userController.register)
10 router.get('/post/add', postController.add)
11 router.post('/post/add', postController.addAction)
12 router.get('/post/:slug/edit', postController.edit)
13
14 module.exports = router
```

Vamos para o postController.js para criarmos a função edit:

```
exports.edit = async (req, res) => {
  const post = await Post.findOne({slug:req.params.slug})
  res.render('postEdit', {post})
}
```

exports.edit = async (req, res) => { → Criação da função edit

1º Passo, precisamos pegar as informações de um post específico

const post = await Post.findOne({slug:req.params.slug}) → Criamos a variável post e adicionamos a função **findOne** que é utilizada para localizar um item específico dentro do banco de dados.

Obs.: Existem diversas formas de acessarmos os itens do banco de dados e nós veremos mais adiante. Aqui utilizaremos a **findOne**.

Iremos utilizar o slug para procurar o post desejado.

{slug:req.params.slug} → O slug receberá da requisição, o parâmetro slug lá da rota no index.js (router.get('/post/:slug/edit', postController.edit)).

res.render('postEdit', {post}) → Carregamos o formulário de edição (res.render)

postEdit → Carregamos o **view** de edição que criaremos logo a seguir com este nome.

{post} → Os dados que carregaremos no formulário de edição.

Obs.: Inicialmente o Bonieky editou essa linha assim: **res.render('postEdit', {post:post})**, mas depois ele explicou que pelo fato de a variável post aqui, ter o mesmo nome do item que queremos editar, podemos excluir uma palavra post daqui, deixando esta linha mais resumida, conforme deixei no código.

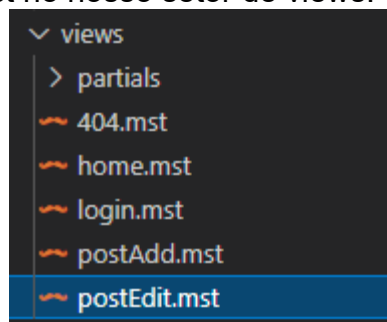
Ficamos com o postController.js assim:

```

controllers > JS postController.js > edit > exports.edit > post
1  const mongoose = require('mongoose')
2  const Post = mongoose.model('Post')
3
4  exports.add = (req, res) => {
5    res.render('postAdd')
6  }
7  exports.addAction = async (req, res) => {
8    const post = new Post (req.body)
9    try{
10     await post.save()
11   } catch(error){
12     req.flash('error', 'Ocorreu um erro! Tente novamente mais tarde!')
13     return res.redirect('/post/add')
14   }
15   req.flash('success', 'Post salvo com sucesso')
16   res.redirect('/')
17 }
18 exports.edit = async (req, res) => {
19   const post = await Post.findOne({slug:req.params.slug})
20   res.render('postEdit', {post})
21 }

```

Agora, iremos criar o arquivo postEdit.mst no nosso setor de views:



Para iniciar, copiamos todo o conteúdo do postAdd.mst, editamos o <h1> para Editar Post e adicionamos os itens do post que será editado. Ficando com o **postEdit.mst** desta forma:

```

views > postEdit.mst
1  {{> header}}
2
3  <h2>Editar Post</h2>
4  <form method="POST">
5    <label>
6      Título:
7      <input type="text" name="title" value="{{post.title}}">
8    </label>
9    <br>
10   <label>
11     Corpo:
12     <textarea name="body">{{post.body}}</textarea>
13   </label>
14   <br>
15   <label>
16     Tags:
17   </label>
18   <br>
19   <input type="submit" value="Salvar">
20 </form>

```

Da cópia que fizemos do addPost para cá, além de editar o texto do <h1> adicionamos o **value="{{post.title}}"** do input title da **linha 7** e adicionamos o conteúdo **{{post.body}}** no testarea da **linha 12**.

A partir daqui, a nossa página de editar post já está funcionando:

Editar Post

Título:

Corpo:

Corpo do post 2.

Tags:

Porém, precisamos agora, criar a rota para salvarmos as alterações que forem realizadas. Para isto voltamos ao index.js e criamos a seguinte rota:

```
router.post('/post/:slug/edit', postController.editAction)
```

router.post('/post/:slug/edit', postController.editAction) → Para salvarmos, utilizamos o método de envio 'post' e utilizamos uma função que criaremos no nosso postController chamado addAction.

Obs.: agora temos add, addAction, edit e editAction

Nosso index.js fica assim:

```

routes > JS index.js > ...
1  const express = require('express')
2  const homeController = require('../controllers/homeController')
3  const userController = require('../controllers/userController')
4  const postController = require('../controllers/postController')
5
6  const router = express.Router()
7  router.get('/', homeController.index)
8  router.get('/users/login', userController.login)
9  router.get('/users/register', userController.register)
10 router.get('/post/add', postController.add)
11 router.post('/post/add', postController.addAction)
12 router.get('/post/:slug/edit', postController.edit)
13 router.post('/post/:slug/edit', postController.editAction)
14
15 module.exports = router

```

O editAction precisará realizar as seguintes funções:

- Procurar o item enviado.
- Pegar os dados e atualizar.
- Mostrar a mensagem de sucesso.
- Redirecionar para a página home.

```

exports.editAction = async (req, res) => {
  const post = await Post.findOneAndUpdate(
    {slug:req.params.slug},
    req.body,
    {
      new:true,
      runValidators:true
    }
  )
  req.flash('success', 'Post atualizado com sucesso!')
  res.redirect('/')
}

```

PROCURA O ITEM ENVIADO:

const post = await Post.findOneAndUpdate() → Função que procura um item e atualiza o mesmo.

.findOneAndUpdate() → Possui três parâmetros:

1º Parâmetro → **{slug:req.params.slug}** → Procura o item.

2º Parâmetro → **req.body** → Recebe os novos dados para este item.

3º Parâmetro → Recebe algumas opções que podemos colocar. Neste caso, utilizaremos duas opções:

PEGAR OS DADOS E ATUALIZAR:

new:true → Este comando faz com que, uma vez feita as alterações, ele retorna o novo post, com as informações atualizadas.

Obs.: Se não adicionarmos o 'new:true' aqui, por padrão, ele retorna as informações do post antigo

runValidators:true → Para aceitar a edição do post, este comando verifica todos os itens de validação existentes no formulário antes de salvar as alterações. Por exemplo, os campos com 'required' ou outros tipos de validações.

MOSTRAR A MENSAGEM DE SUCESSO:

`req.flash('success', 'Post atualizado com sucesso!')` → Utilizamos uma mensagem flash para informar o usuário que a edição foi feita com sucesso.

REDIRECIONAR PARA HOME:

`res.redirect('/')`

Navegador:

Post atualizado com sucesso!

Página Home

[Título do Post 1.1](#)

Corpo do Post 1.1

[\[Editar \]](#)

Aula 10 – Otimizando o Editar

PRIMEIRA OTIMIZAÇÃO:

findOneAndUpdate → O mongoose já utilizava esta função, porém, quando ele acessava o Mongo DB, ele utilizava a função `findAndModify`. A função `findAndModify`, está depreciada pelo Mongo DB, por este motivo, precisamos desativar esta função para que o mongoose utilize sempre a `findOneAndUpdate` e não tenha conflitos ao acessar o Mongo DB.

findAndModify → função que foi depreciada pelo mongo DB

Para desativarmos a `findAndModify`, vamos acessar o nosso arquivo **server.js** e nas configurações do **mongoose.connect** adicionamos o seguinte comando: **useFindAndModify:false**

Nosso `server.js` fica assim:

```
JS server.js > useFindAndModify
1  const mongoose = require('mongoose')
2  require('dotenv').config({path: 'variables.env'})
3
4  //CONEXÃO AO BANCO DE DADOS
5  mongoose.connect(process.env.DATABASE, {
6    useUnifiedTopology: true,
7    useFindAndModify: false
8  })
9  mongoose.Promise = global.Promise
10 mongoose.connection.on('error', (error)=> {
11   console.error("ERRO: "+error.message)
12 })
13
14 //Carregando todos os models
15 require('./models/Post')
16 const app = require('./app')
17
18 app.set('port', process.env.PORT || 7777)
19 const server = app.listen(app.get('port'), ()=>{
20   console.log("Servidor rodando na porta"+server.address().port)
21 })
```

SEGUNDA OTIMIZAÇÃO:

Outra observação é que **ao atualizarmos um post, o slug do post não está sendo alterado**, mesmo que o seu título tenha sido alterado. Isto acontece porque a função `postSchema.pre` que está no `Post.js` não está sendo executada durante o processo de edição do post, esta função só está sendo executada no processo de criação.

Para que possamos atualizar o slug de um post que tenha o seu título alterado, podemos fazer da seguinte maneira:

Acessamos o `postController.js` e adicionamos a seguinte linha dentro do `editAction`:

```
exports.editAction = async (req, res) => {  
  req.body.slug = require('slug')(req.body.title, {lower:true})
```

`req.body.slug = require('slug')(req.body.title, {lower:true})` → Cria a requisição do slug e roda como uma função e enviamos o título para o novo slug e forçamos a utilização de todas as letras em minúsculas.

Outra opção de fazer o mesmo processo (Obs.: esta segunda foi a opção que o Bonieky utilizou) é: Lá no início do `postController`, criamos uma variável e fazemos a requisição do slug, desta forma:

```
const slug = require('slug')
```

```
const mongoose = require('mongoose')  
const slug = require('slug')  
const Post = mongoose.model('Post')
```

E lá dentro do `editAction` modificamos a linha que adicionamos anteriormente deixando ela desta forma:

```
exports.editAction = async (req, res) => {  
  req.body.slug = slug(req.body.title, {lower:true})
```

```
exports.editAction = async (req, res) => {  
  req.body.slug = slug(req.body.title, {lower:true})  
  const post = await Post.findOneAndUpdate(  
    {slug:req.params.slug,  
    req.body,  
    {  
      new:true,  
      runValidators:true  
    }  
  )  
  req.flash('success', 'Post atualizado com sucesso!')  
  res.redirect('/')  
}
```

TERCEIRA OTIMIZAÇÃO:

try catch no edit:

Trazendo o conjunto de comandos **try catch** aqui para o nosso `editAction`, podemos trabalhar com alguns possíveis erros que possam vir a acontecer. Editamos o `editAction` desta forma:

```
exports.editAction = async (req, res) => {
  req.body.slug = slug(req.body.title, {lower:true})
  try {
    const post = await Post.findOneAndUpdate(
      {slug:req.params.slug},
      req.body,
      {
        new:true,
        runValidators:true
      }
    )
  } catch(error) {
    req.flash('error', 'Ocorreu um erro! Tente novamente mais tarde!')
    return res.redirect(['/post/'+req.params.slug+'/edit'])
  }
  req.flash('success', 'Post atualizado com sucesso!')
  res.redirect('/')
}
```

Adicionamos o **try** para a tentativa de edição.

Adicionamos o **catch** após o bloco de comandos do try, exatamente como fizemos no addAction. Inclusive, copiamos essas duas linhas do catch lá do addAction e alteramos apenas o endereço para o redirect

'/post/'+req.params.slug+'/edit' → Aqui, redirecionamos para a própria página de edição de post com o post a ser editado preenchido

Aula 11 – Adicionar Tags

Para adicionar as Tags, primeiramente, criamos este campo no postAdd.mst da seguinte forma:

```
<label>
  Tags: <small>(separar as tags com vírgulas)</small><br>
  <input type="text" name="tags">
</label>
```

Como utilizamos aqui no name o mesmo nome que temos lá no item do nosso schema, se adicionarmos alguma informação neste input, ele encontrará automaticamente lá no banco de dados o item de nome tags e salvará a informação recebida.

```
const postSchema = new mongoose.Schema({
  title:{
    type:String,
    trim:true,
    required:'O post precisa de um título'
  },
  slug:String,
  body:{
    type:String,
    trim:true
  },
  tags:[String]
})
```

Mas, se salvarmos diversas tags, separadas por vírgulas, de acordo com a orientação, lá no banco de dados, teremos um array com todas as tags adicionada em uma única posição do array, e não é isso exatamente que queremos.

Adicionar Post

Título:

Corpo:

Tags: (separar as tags com vírgulas)

```
{
  _id: ObjectId("60b9562dfe5e84318cc5e3b8")
  tags: Array
    0: "post, 3, corpo, do"
  title: "Post 3"
  body: "Corpo do post3"
  slug: "post-3"
  __v: 0
}
```

Para corrigir este problema, vamos adicionar a seguinte linha no nosso addAction do arquivo postController.js:

req.body.tags = req.body.tags.split(',').map(t=>t.trim()) → Realizamos a requisição do campo tags, utilizamos o comando **split** e separamos o texto recebido em todo o lugar onde for encontrado vírgula. Colocamos cada uma destes itens em uma posição do array. Com o **.map**, mapeamos (percorremos) o array utilizando o comando **trim** que remove os espaços excedentes no início e no final de cada item.

Adicionar Post

Título:

Corpo:

Tags: (separar as tags com vírgulas)

```
{
  _id: ObjectId("60b96ae8d5bdc7339427e709")
  tags: Array
    0: "post"
    1: "corpo"
    2: "4"
    3: "do"
  title: "post 4"
  body: "Corpo do post 4"
  slug: "post-4"
  __v: 0
}
```

addAction ficou assim:

```
exports.addAction = async (req, res) => {
  req.body.tags = req.body.tags.split(',').map(t=>t.trim())
  const post = new Post (req.body)
  try{
    await post.save()
  } catch(error){
    req.flash('error', 'Ocorreu um erro! Tente novamente mais tarde!')
    return res.redirect('/post/add')
  }
  req.flash('success', 'Post salvo com sucesso')
  res.redirect('/')
}
```

Aula 12 – Editar Tags

Para editarmos as tags, a primeira coisa que precisamos fazer é adicionar o campo tags na tela de edição do post.

Para isto, adicionaremos os seguintes comandos no nosso editPost.mst:

```
<label>
  Tags: <small>(separar as tags com vírgulas)</small><br>
  <input type="text" name="tags" value="{{post.tags}}">
</label>
```

Quando adicionamos o **value="{{post.tags}}"** o mustache se encarrega de pegar o array que temos no nosso banco de dados e transforma em um array para ser apresentado neste input tipo texto.

Após esta edição, precisamos apenas adicionar uma linha no editAction do postController.js:

```
exports.editAction = async (req, res) => {
  req.body.slug = slug(req.body.title, {lower:true})
  req.body.tags = req.body.tags.split(',').map(t=>t.trim())
```

req.body.tags = req.body.tags.split(',').map(t=>t.trim()) → A mesma linha de comandos que adicionamos no addAction para tratar as tags após a edição. Utilizamos ela aqui para serem salvas no banco de dados no mesmo formato em que são adicionadas pela primeira vez.

Aula 13 – Visualizar Post Único

Em primeiro lugar, vamos criar a rota para a página de visualização de um post único. Para isto acessamos o index.js e criamos a rota:

```
router.get('/post/:slug', postController.view)
```

E ficamos com o index assim:

```
routes > JS index.js > [?] <unknown>
1  const express = require('express')
2  const homeController = require('../controllers/homeController')
3  const userController = require('../controllers/userController')
4  const postController = require('../controllers/postController')
5
6  const router = express.Router()
7  router.get('/', homeController.index)
8  router.get('/users/login', userController.login)
9  router.get('/users/register', userController.register)
10
11 router.get('/post/add', postController.add)
12 router.post('/post/add', postController.addAction)
13
14 router.get('/post/:slug/edit', postController.edit)
15 router.post('/post/:slug/edit', postController.editAction)
16
17 router.get('/post/:slug', postController.view)
18
19 module.exports = router
```

Aqui, sobre a ordem das rotas criadas, o Bonieky fez uma observação **muito importante**: Esta última rota que criamos deve ficar em último lugar nesta lista para que não haja conflito de rotas no nosso sistema. Isto ocorre primeiramente porque o javascript executa seus comandos em ordem, ou seja, linha a linha.

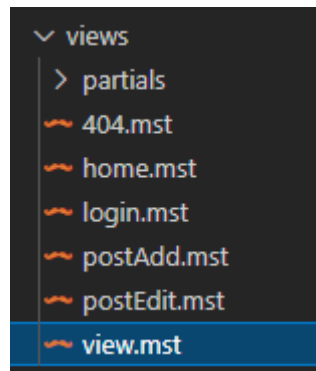
Se tivéssemos adicionado a rota **router.get('/post/:slug', postController.view)** acima das outras, e na nossa página clicássemos em adicionar post o javascript executaria a linha de visualizar um post, pois as duas rotas começam com a palavra 'post' e o ':slug' da rota view aceitaria a palavra 'add' da rota de adicionar post e trataria a palavra 'add' como um slug.

Para que isso não aconteça, colocamos a rota de visualização dos posts por último. E para que também não houvesse conflito com as rotas de editar os posts.

Agora que criamos a rota retornamos ao postController.js para criarmos o **.view** que não temos ainda:

```
exports.view = (req, res) => {  
  res.render('view')  
}
```

Utilizamos aqui o **res.render('view')** apenas para teste. Agora precisamos criar o arquivo view dentro do **views**



Com o view.mst criado, podemos configurar o export.view no postController.js desta forma:

```
exports.view = async (req, res) => {  
  const post = await Post.findOne({slug:req.params.slug})  
  res.render('postEdit', {post})  
}
```

Vamos editar o view.mst para que exiba um único post específico:

```
views > view.mst  
1  {{> header}}  
2  
3  <h1>{{post.title}}</h1>  
4  <article>{{post.body}}</article>  
5  <hr>  
6  <small>Tags: {{post.tags}}</small>
```

Após criado o view.mst conforme a imagem acima, temos no navegador o seguinte resultado:

Cabeçalho

- [Home](#)
 - [Login](#)
 - [Adicionar Post](#)
-

Post 3

Corpo do post3

Tags: post, 3, corpo, do

Aula 14 – Inserindo CSS no Template

O primeiro passo que vamos dar será criar a estrutura de pastas que precisaremos para trabalhar com a estilização do projeto.

Criaremos uma pasta pública para o nosso projeto onde poderemos inserir imagens do próprio sistema, do próprio site, CSS ou qualquer outro arquivo externo que quisermos adicionar

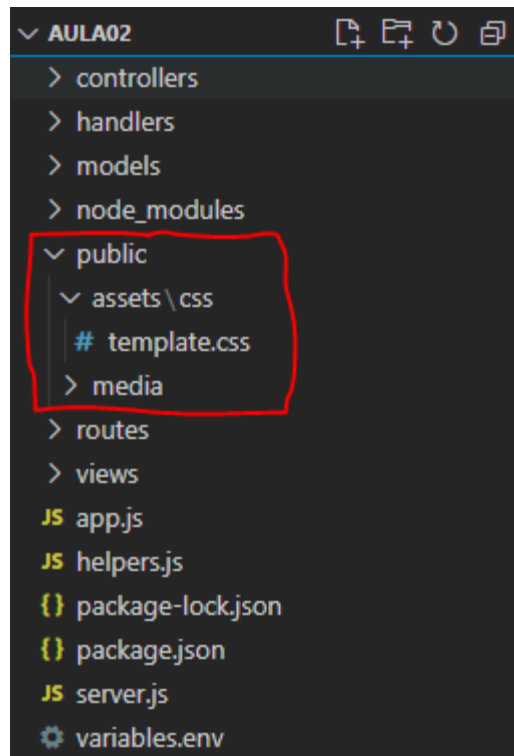
Para isto criaremos uma pasta na raiz do nosso projeto chamado de '**public**'. Geralmente, todo o projeto possui uma pasta public.

Dentro da pasta public criaremos outras duas pastas:

assets → Será para colocarmos os arquivos do sistema, como ícones, arquivos de css e outros.

media → Para receber arquivos externos recebidos por uploads.

Dentro da pasta assets, criaremos uma pasta chamada de 'css' e dentro desta pasta, criaremos um arquivo chamado de **template.css**



A partir daqui, faremos com que essa pasta public fique acessível. Para isto, primeiramente vamos editar o nosso app.js.

Obs.: As configurações a seguir devem ser colocadas antes das rotas do app.js, para que antes de acessar qualquer URL seja verificado se existe algum arquivo na pasta public antes de procurar por alguma rota.

No app.js adicionamos o seguinte comando na linha 14:

```
app.use(express.static(__dirname+'/public'))
```

app.use(express.static(__dirname+'/public')) → Configura a pasta public como uma pasta estática (pública para todo o sistema).

App.js ficou assim:

```
JS app.js > ...
1  const express = require('express')
2  const mongoose = require('mongoose')
3  const mustache = require('mustache-express')
4  const cookieParser = require('cookie-parser')
5  const session = require('express-session')
6  const flash = require('express-flash')
7  const router = require('./routes/index')
8  const helpers = require('./helpers')
9  const errorHandler = require('./handlers/errorHandler')
10 //Configurações
11 const app = express()
12 app.use(express.json())
13 app.use(express.urlencoded({extended:true}))
14 app.use(express.static(__dirname+'/public'))
15 app.use(cookieParser(process.env.SECRET))
16 app.use(session({
17   secret:process.env.SECRET,
18   resave:false,
19   saveUninitialized:false
20 }))
21 app.use(flash())
22 app.use((req, res, next)=>{
23   res.locals.h = helpers
24   res.locals.flashes = req.flash()
```

Agora que habilitamos a pasta public no nosso sistema, podemos utilizar todo o conteúdo que está dentro dela.

Vamos agora utilizar o template.css no nosso header.mst e no home.mst.

header.mst:

<link rel="stylesheet" href="/assets/css/template.css"> → É importante observar aqui, que pelo fato de termos configurado a nossa pasta public como uma pasta pública lá no app.js, não precisamos adicioná-la aqui na declaração do caminho do template.css, pois a pasta public está disponível, acessível e visível a todo o nosso sistema.

<header> → Adicionamos a tag <header> para aplicarmos o css do tamplate.css no Cabeçalho

class="warning" → Adicionamos a tag <div class="warning"> para também aplicarmos o css do tamplate.css

```

views > partials > header.mst
1  <html>
2  <head>
3      <title>{{pageTitle}} {{h.defaultPageTitle}}</title>
4      <link rel="stylesheet" href="/assets/css/template.css">
5  </head>
6  <body>
7  <header>
8      <h1>Cabeçalho</h1>
9      <hr>
10     <ul>
11         {{#h.menu}}
12         <li><a href="{{slug}}">{{name}}</a></li>
13         {{/h.menu}}
14     </ul>
15 </header>
16 {{#flashes}}
17     {{#success}}
18     <div class="warning">{{.}}</div>
19     {{/success}}
20     {{#error}}
21     <div class="warning">{{.}}</div>
22     {{/error}}
23     {{#info}}
24     <div class="warning">{{.}}</div>
25     {{/info}}
26 {{/flashes}}
27 </body>

```

home.mst:

Aqui, no home.mst adicionamos uma <div> e aplicamos a classe post do template.css

```

views > home.mst
1  {{>header}}
2  <h1>Página Home</h1>
3  {{#posts}}
4      <div class="post">
5          <h3><a href="/post/{{slug}}">{{title}}</a></h3>
6          <p>{{body}}</p>
7          <a href="/post/{{slug}}/edit">[ Editar ]</a>
8      </div>
9  {{/posts}}

```

Com a aplicação do css no nosso projeto, as páginas ficaram com as seguintes aparências:

Cabeçalho

Home

Login

Adicionar Post

Página Home

Titulo do Post 1.2

Corpo do Post 1.1

[Editar]

Titulo do Post 2.1

Corpo do post 2.1.

[Editar]

Post 3

Corpo do post3

[Editar]

post 4

Corpo do post 4

[Editar]

Cabeçalho

Home

Login

Adicionar Post

Adicionar Post

Título:

Corpo:

Tags: (separar as tags com vírgulas)

Salvar

Cabeçalho

Home

Login

Adicionar Post

Titulo do Post 1.2

Corpo do Post 1.1

Tags:

Cabeçalho

Home

Login

Adicionar Post

Editar Post

Título:

Titulo do Post 1.2

Corpo:

Corpo do Post 1.1

Tags: (separar as tags com vírgulas)

Salvar