

NodeJS – Módulo 02

Material de estudo desenvolvido por:
Daniel Teixeira Quadros

Sumário

Aula 01 – Variáveis de Projeto.....	2
Aula 02 – Configurando o MongoDB (Banco de Dados).....	4
Aula 03 – Entendendo as Rotas (1/2).....	8
Aula 04 – Entendendo as Rotas (2/2).....	10
Aula 05 – Configurando Template Engine.....	12
Aula 06 – Entendendo Templates (1/2).....	14
Aula 07 – Entendendo Templates (2/2).....	17
Aula 08 – Templates Helpers.....	21
Aula 09 – Estrutura MVC no Node.....	26
Aula 10 – Middleware.....	29

Aula 01 – Variáveis de Projeto

Aqui, por enquanto, prosseguiremos com o projeto que iniciamos no módulo1.

Primeiramente, precisamos criar o que é chamado de variáveis de ambiente (environment variables). As variáveis de ambiente são aquelas variáveis que criaremos com as informações padrões como: Que ambiente do nosso projeto estaremos (Desenvolvimento, Produção, etc...) a url do projeto e outras. Para isto, criamos um novo arquivo no nosso projeto e o chamamos de '**variables.env**' (o .env é de environment (ambiente)).

A estrutura padrão para este tipo de arquivo é: **NOME=valor**

Utilizaremos as seguintes configurações:

NODE_ENV=desenvolvimento → Variável de ambiente, utilizamos o conteúdo development para indicar que estamos em ambiente de desenvolvimento.

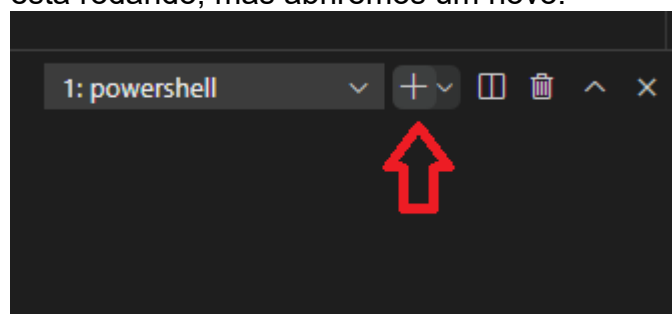
- Podemos colocar aqui por exemplo, a url do MongoDB de produção e a url de local. Aí, dependendo da variável de ambiente podemos conectar em um ou em outro.

PORT=7777 → Porta, utilizaremos a padrão '7777', aí se quisermos mudar a porta, é só alterarmos este valor e o servidor mudará a porta.

DATABASE=mongodb://127.0.0.1:27017 → Configurações para conexão com o Banco de Dados. Como saber esta configuração? A Configuração do servidor local é muito simples de saber. Para isto utilizaremos o terminal:

Obs, se tivesse usuário e senha, utilizaríamos aqui 'at @ senha'.

Obs.: Não utilizaremos o que está rodando, mas abriremos um novo:



Nesta etapa aqui, por algum motivo que eu não descobri, não funcionou aqui. Mas as etapas que ele orientou foram:

Após abrir um novo terminal, como na figura acima, que na tela dele ficou assim:



Após ele dar 'Enter' ficou assim:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: mongo
2019-07-31T10:41:26.495-0300 I CONTROL [initandlisten]
-----
Enable MongoDB's free cloud-based monitoring service, which will then receive and display
metrics about your deployment (disk utilization, CPU, operation statistics, etc).

The monitoring data will be available on a MongoDB website with a unique URL accessible to you
and anyone you share the URL with. MongoDB may use this information to make product
improvements and to suggest MongoDB products and deployment options to you.

To enable free monitoring, run the following command: db.enableFreeMonitoring()
To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
-----
```

E no meu

deu erro de comando desconhecido.

No terminal dele, ele rolou a tela um pouco para cima e encontrou a seguinte informação '**connecting to:**', que foi utilizada na variável DATABASE, → **mongodb://127.0.0.1:27017**

Obs. mesmo sem conseguir realizar esta etapa, copieei esta configuração e adicionei na minha variável DATABASE.

Além de especificar o servidor, na variável DATABASE, precisaremos configurar também o Banco de Dados. Que criaremos mais adiante.

Por enquanto, o nosso arquivo variables.env ficou assim:

```
aula02 > variables.env
1  NODE_ENV=developmente
2  PORT=7777
3  DATABASE=mongodb://127.0.0.1:27017
```

Agora, precisamos fazer com que o nosso servidor leia as variáveis de ambiente e utilize elas. Para isto retornamos ao nosso arquivo **server.js** e instalamos uma extensão chamada **dotenv** que é utilizada para isso.

No terminal, na pasta do nosso projeto, executamos o comando **npm install dotenv**:

Obtivemos o seguinte resultado:

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

> aula02@1.0.0 start D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02
> nodemon ./server.js

[nodemon] 2.0.7
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] starting `node ./server.js`
Servidor rodando na porta7777
[nodemon] restarting due to changes...
[nodemon] restarting due to changes...
[nodemon] starting `node ./server.js`
Servidor rodando na porta7777
Terminate batch job (Y/N)? n
PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02> npm install dotenv
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"})

+ dotenv@9.0.2
added 1 package and audited 169 packages in 4.103s

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

found 0 vulnerabilities

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

Agora podemos utilizar esta extensão:

No server.js →

Vamos utilizar um require direto na extensão, para já utilizar, sem precisar criar uma constante, ou nada disso.

```
require('dotenv').config({path: 'variables.env'})
```

E já passamos para ela, como um objeto no 'path:', a **localização** do nosso arquivo que contém as variáveis de ambiente.

Obs.: A partir daqui, já podemos utilizar as nossas variáveis de ambiente.

Agora, ao invés de, adicionarmos o número da porta do nosso servidor diretamente aqui no arquivo server.js, **podemos puxar esta informação do 'variables.env'** que foi criado exatamente para conter essas informações de configuração.

Aqui, trocamos a informação do número da Porta '**app.set('port', 7777)**' pelo comando '**app.set('port', process.env.PORT || 7777)**' que busca no nosso arquivo com extensão **.env** a variável de nome PORT (Aquele que criamos lá no arquivo 'variables.env') e, por segurança adicionamos o (**|| 7777**). Que, se por acaso, não encontrar a variável PORT, utilizará a porta '7777'.

Terminamos a aula com o server.js assim:

Aula 02 – Configurando o MongoDB (Banco de Dados)

Agora, já temos rodando o nosso projeto (servidor) e o nosso banco de dados. Precisamos agora interligar o nosso banco de dados ao nosso projeto.

No Mongo DB Compass, dentro do servidor que criamos, iremos criar o banco de dados. Para isto clicamos em 'CREATE DATABASE' e aparecerá a seguinte tela:

Create Database

Database Name

Collection Name

☐ Capped Collection ⓘ

☐ Use Custom Collation ⓘ

Before MongoDB can save your new database, a collection name must also be specified at the time of creation. [More Information](#)

CANCEL CREATE DATABASE

Aqui, estamos utilizando o exemplo de um Blog, e por isto, o Bonieky fez a escolha dos 'Names' de acordo com a característica do projeto.

Database Name →

Collection Name →

Create Database

Database Name

blog

Collection Name

posts

☐ Capped Collection ⓘ

☐ Use Custom Collation ⓘ

Before MongoDB can save your new database, a collection name must also be specified at the time of creation. [More Information](#)

CANCEL CREATE DATABASE





Aqui preenchemos com as seguintes configurações e clicamos em 'CREATE DATABASE':

O nosso Banco de Dados já aparece aqui na lista:


Databases

Performance

CREATE DATABASE

Database Name ^	Storage Size	Collections	Indexes	
admin	20.0KB	0	1	
blog	4.0KB	1	1	
config	24.0KB	0	2	
local	36.0KB	1	1	

Clicando no nosso Banco de Dados, já aparece a nossa collection 'posts':

CREATE COLLECTION						
Collection Name ^	Documents	Avg. Document Size	Total Document Size	Num. Indexes	Total Index Size	Properties
posts	0	-	0.0 B	1	4.0 KB	

Para fazermos a conexão do Banco de Dados com o nosso sistema, utilizaremos uma dependência chamado 'mongoose'. Para isto, executamos o seguinte comando no terminal:

npm install mongoose --save

```
PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1> cd aula02
PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02> npm install mongoose --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"})

+ mongoose@5.12.9
added 29 packages from 92 contributors and audited 198 packages in 7.132s

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

found 0 vulnerabilities

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

Acessamos o 'variables.env' para adicionar a collection 'blog' no DATABASE:

```
aula02 > variables.env
1  NODE_ENV=desenvolvimento
2  PORT=7777
3  DATABASE=mongodb://127.0.0.1:27017/blog
```

Após adicionarmos a nossa collection, voltamos para o nosso arquivo server.js para conectarmos o mongoose a ele:

const mongoose = require('mongoose') → Adicionamos as propriedades do mongoose em uma 'const' que colocamos o nome de mongoose.

mongoose.connect(process.env.DATABASE) → Comando para conectar o nosso mongoose utilizando a **string de conexão**. A nossa string de conexão está na nossa variável de ambiente **dentro do arquivo 'variables.env' com o nome DATABASE**

Obs.: Com isto já é o suficiente para ele conectar, porém precisamos configurar algumas coisinhas a mais:

mongoose.Promise = global.Promise → Esta linha de comando informa para o mongoose (torna compatível) que ele pode usar ECMASCRIPT 6 dentro das conexões com o banco de dados.

```
mongoose.connection.on('error', (error)=> {  
  console.error("ERRO: "+error.message)  
})
```

Este último comando, serve para que, em caso de acontecer algum erro, possamos mostrar este erro no nosso Log.

Este comando possui dois parâmetros:

1º - Informamos o que estamos monitorando. Neste caso, são os erros ('error').

2º - É uma função anônima que recebe o próprio erro e exibimos ele. Poderíamos utilizar o console.log, porém utilizamos o **console.error** que mostra este erro de uma forma diferente.

Terminando esta etapa, voltamos para o **terminal do node** para verificarmos se houve algum erro. Neste caso, não houve erro, mas apenas um aviso.

Neste caso, vamos acatar ao aviso que está dizendo que o **url parser** que é o padrão do connect, ele vai mudar e vai ser removido:

```
(node:12700) DeprecationWarning: current URL string parser is deprecated, and will be removed in a future version. To use the new parser, pass option { useNewUrlParser: true } to MongoClient.connect.  
(Use `node --trace-deprecation ...` to show where the warning was created)  
(node:12700) [MONGODB DRIVER] Warning: Current Server Discovery and Monitoring engine is deprecated, and will be removed in a future version. To use the new Server Discover and Monitoring engine, pass option { useUnifiedTopology: true } to the MongoClient constructor.  
Servidor rodando na porta7777
```

Então, iremos utilizar esta opção, { useUnifiedTopology: true }. Isto é um processo interno do mongoose.

Para isto, copiamos este objeto ({ useUnifiedTopology: true }) e colocamos no segundo parâmetro do connect, ficando assim:

```
mongoose.connect(process.env.DATABASE, { useUnifiedTopology: true })
```

```
MODULO1
└─ aula02
   ├── node_modules
   ├── JS app.js
   ├── {} package-lock.json
   ├── {} package.json
   └─ JS server.js
      └─ variables.env

aula02 > JS server.js > ...
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  app.set('port', process.env.PORT || 7777)
12  const server = app.listen(app.get('port'), ()=>{
13    console.log("Servidor rodando na porta"+server.address().port)
14  })
```

Aula 03 – Entendendo as Rotas (1/2)

Com o desenvolvimento do nosso sistema, acabamos tendo que criar diversas rotas. E para uma melhor organização do nosso sistema, separamos esta estrutura em arquivos diferentes.

Um conceito importante aqui, é que podemos estabelecer rotas específicas para o site e rotas específicas para o painel de administração. Ambas formam, ambientes como se fossem sistemas separados.

Uma forma de trabalharmos desta maneira é criarmos **uma pasta dentro do nosso sistema chamado de 'routes'**. E neste exercício, dentro desta pasta, criamos os arquivos **'index.js'** (para o site, sistema). Poderíamos ter criado o arquivo **'admin.js'** (para o administrador), mas não utilizamos aqui.

app.use('/admin', adminRouter) → Este comando criaria uma segunda rota que utilizaríamos para o administrador do site. Porém, não utilizaremos esta linha agora. Aqui, já havíamos criado a rota para o usuário, a **app.use('/', router)**.

Para utilizarmos esta estrutura, tiraremos as linhas com os comandos daqui (app.js):

```
const router = express.Router()
router.get('/', (req, res)=>{
  res.send('Olá Mundo!')
})
```

daqui, e passaremos para o **'index.js'**.

Dentro do **index.js**, antes destes comando, precisamos adicionar o express (**const express = require('express')**), para que haja a conexão, e **ao final** adicionamos ainda o **'module.exports = router'**

Depois de ter removido do app.js para o index.js, as linhas:

```
const router = express.Router()
router.get('/', (req, res)=>{
  res.send('Olá Mundo!')
})
```

Adicionamos o seguinte comando:

```
const router = require('./routes/index') // Que importará a rota index para cá
```

O nosso app.js ficou assim:


```

aula02 > JS app.js > ...
1  const express = require('express')
2  const router = require('./routes/index')
3
4  //Configurações
5  const app = express()
6  app.use('/', router)
7  module.exports = app

```

No arquivo index.js, criaremos uma outra rota para exercitarmos:

```

router.get('/sobre', (req, res)=>{
  res.send('Página SOBRE')
})

```

E já está ativa, podemos testá-la acessando o navegador e digitando: <http://localhost:7777/sobre>

Nosso arquivo index.js ficou assim:

```

1  const express = require('express')
2
3  const router = express.Router()
4  router.get('/', (req, res)=>{
5    res.send('Olá Mundo!')
6  })
7
8  router.get('/sobre', (req, res)=>{
9    res.send('Página SOBRE')
10 })
11
12 module.exports = router

```

E a estrutura no VSCode, ficou assim:

```

v aula02
  > node_modules
  v routes
  JS index.js
  JS app.js
  {} package-lock.json
  {} package.json
  JS server.js
  ⚙ variables.env

```

Aula 04 – Entendendo as Rotas (2/2)

router.get('/', (req, res)=>{

req → Aqui no 'req' terão todas as informações relacionadas a sua requisição: ou seja, cabeçalho de requisição, parâmetros específicos que forem enviados, se o usuário fez algum upload de arquivo, todos estes detalhes estarão aqui no req (tudo do acesso)

Como acessamos esses conteúdos?

VIA GET → Na barra de endereços utilizaremos uma 'query string (É quando enviamos alguns dados via método GET)' Ex.:

http://localhost:7777/?nome=Daniel&idade=40, neste exemplo, embora não tenha ocorrido nenhuma alteração em nossa página ou site, eles estão disponíveis no 'req'

E para acessá-los

Aqui utilizamos o send. Para visualizarmos:

```
const router = express.Router()
  let nome = req.query.nome
  let idade = req.query.idade

  res.send('Olá '+nome+' Você tem '+idade+ ' anos de idade')
})
```

VIA POST → Primeiramente temos que fazer uma adição no nosso app.js

```
aula02 > JS app.js > ...
1  const express = require('express')
2  const router = require('./routes/index')
3
4  //Configurações
5  const app = express()
6  app.use('/', router)
7
8  app.use(express.json())
9
10 module.exports = app
```

app.use(express.json()) → **express.json** → Recurso do Express, que faz com que os conteúdos do POST sejam tratados da mesma forma que são tratadas as requisições GET. Utilizará um objeto tipo json.

Obs.: Mais adiante verificaremos como enviar estes POST.

Veremos **outro recurso do 'req'** que trata de, como pegar parâmetros específicos na rota. Para isto criaremos uma outra rota para testes, utilizando o nosso exercício com a ideia de um blog, vamos criar uma rota que pegaria um post(postagem) específica dentre os posts do blog:

router.get('/posts/:id', (req, res)=>{ → Aqui, o '**posts**' dá a ideia da rota dos posts e o **:id** é utilizado para pegarmos um post com um id específico que será vinculado a ele.

let id = req.params.id → Para acessarmos este id, utilizamos esta linha onde o 'params', indica os parâmetros que estão sendo enviados aqui, e um deles é o id

res.send('ID do post: '+id) → Utilizamos esta linha para verificar se esta rota está funcionando.

Para testarmos esta rota, utilizamos na barra de endereços do navegador, o seguinte endereço:

http://localhost:7777/posts/1 e obtivemos a resposta na tela. Conforme eu mudar o número final deste endereço, o site responde de acordo com o número adicionado, como se fosse o número de um post específico mesmo.

VEREMOS AGORA UM POUCO SOBRE O PARÂMETRO 'res' RESPOSTA:

```
router.get('/', (req, res)=>{
  let nome = req.query.nome
  let sobrenome = req.query.sobrenome
  res.json({
    nomeCompleto: nome+' '+sobrenome
  })
})
```

res.json({nomeCompleto: nome+' '+sobrenome}) → É a forma usual de enviar objetos.

Obs.: Aqui, não utilizamos o '.send', já que estamos enviando via json.

Para testarmos, utilizamos o seguinte endereço na barra de endereços: **http://localhost:7777/?nome=Daniel&sobrenome=Quadros** e obtivemos o seguinte resultado na nossa página: **{"nomeCompleto":"Daniel Quadros"}**

Obs.: Esta resposta na tela tem exatamente o formato que um servidor precisa.

Um outro exemplo aqui é, de tudo o que for mandado, será transformado em json:

res.json(req.query) → Este comando pega a requisição inteira e envia como resposta. Desta forma, tudo o que enviarmos, será transformado em json. Utilizamos o endereço **http://localhost:7777/?nome=Daniel&sobrenome=Quadros&idade=40** no navegador e obtivemos o retorno **{"nome":"Daniel","sobrenome":"Quadros","idade":"40"}** na tela.

Desta forma, podemos enviar um array, ou fazermos o que quiser com este item aqui.

```
router.get('/', (req, res)=>{
  res.json(req.query)
})
```

Obs.: Aqui, em todos estes exemplos, utilizamos o método GET

GET: req.query

POST: req.body

PARÂMETROS DA URL: req.params

ENVIO:

SEND

JSON

Nosso index.js ficou assim:

```
aula02 > routes > JS index.js > ...
1  const express = require('express')
2
3  const router = express.Router()
4  router.get('/', (req, res)=>{
5    res.json(req.query)
6  })
7
8  router.get('/:id', (req, res)=>{
9    let id = req.params.id
10    res.send('ID do post: '+id)
11  })
12
13  router.get('/sobre', (req, res)=>{
14    res.send('Página SOBRE')
15  })
16
17  module.exports = router
```

Aula 05 – Configurando Template Engine

Para iniciarmos esta aula, começamos fazendo uma limpeza no nosso `index.js`, deixando ele assim:

```
aula02 > routes > JS index.js > router.get('/') callback
1  const express = require('express')
2
3  const router = express.Router()
4  router.get('/', (req, res) => {
5
6  })
7
8  module.exports = router
```

Template Engine → Aqui utilizaremos o **Mustache**. Instalaremos o **Mustache para o Express**.

Para isto, utilizamos o comando `'npm install mustache-express --save'` no terminal dentro da pasta do projeto:

```
PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1> cd aula02
PS D:\Informática\Programação\2021\Daniel\B7Web\NodeJS\modulo1\aula02> npm install mustache-express --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"})

+ mustache-express@1.3.0
added 5 packages from 11 contributors and audited 203 packages in 8.776s

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

found 0 vulnerabilities

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

Configuraremos o Mustache no arquivo **app.js** que é onde configuramos todas as configurações da nossa aplicação.

Trazemos a instalação para o nosso projeto através de um `'require'`:

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

Para utilizarmos o Mustache para configurar, executamos o seguinte comando:

```
app.engine('mst', mustache())
```

engine → para especificar qual o motor iremos utilizar

'mst' → Iremos chamar de **mst** (abreviação de Mustache)

mustache() → Iremos rodar a nossa **const mustache** como uma função. Obs.: Mais para frente utilizaremos alguns parâmetros específicos que esta função tem, porém, por enquanto não utilizaremos nenhum parâmetro.

Abaixo, configuramos este motor de visualização da seguinte forma:

```
app.set('view engine', 'mst')
```

Agora, precisamos preparar um local para colocar estes arquivos visuais trazidos pelo Mustache. Para isto, criaremos uma pasta que chamamos de **'views'** na raiz do projeto

E precisamos especificar esta pasta no nosso servidor, no arquivo **app.js** mesmo com o seguinte comando:

```
app.set('views', __dirname + '/views')
```

onde o primeiro parâmetro, 'views' especifica o tipo de arquivos e o segundo, __dirname + '/views' mapeia o local (endereço).

Por enquanto, o nosso app.js ficou assim:

```
aula02 > JS app.js > ...
1  const express = require('express')
2  const mustache = require('mustache-express')
3  const router = require('./routes/index')
4
5  //Configurações
6  const app = express()
7  app.use('/', router)
8
9  app.use(express.json())
10
11 app.engine('mst', mustache())
12 app.set('view engine', 'mst')
13 app.set('views', __dirname + '/views')
14
15 module.exports = app
```

A partir daqui, prosseguimos para o nosso arquivo **index.js** para configurar a rota:

na rota raiz da nossa aplicação, ao invés de utilizarmos um **.send** ou um **.json** utilizaremos o **.render** de renderizar:

```
res.render('home')
```

O render tem dois parâmetros, sendo que apenas o primeiro é obrigatório.

1º – O nome do arquivo que iremos renderizar que precisa estar na pasta views. Que aqui, chamamos de 'home'.

2º – Vão os dados que enviaremos para o view, mas por enquanto, não iremos utilizar.

```
aula02 > routes > JS index.js > <unknown>
1  const express = require('express')
2
3  const router = express.Router()
4  router.get('/', (req, res)=>{
5    res.render('home')
6  })
7
8  module.exports = router
```

Agora, quando acessarmos a '/' no servidor, ele vai executar o **.render('home')**.

Vamos para a pasta views para criar este **home**.

Na pasta views, criamos um arquivo chamado **home.mst** e para testarmos, utilizamos duas linhas de comandos html para podermos visualizá-lo no navegador:

```
aula02 > views > home.mst
1  <h1>Seja bem vindo</h1>
2  <p>Qualquer mensagem</p>
```

Agora, utilizaremos o **segundo parâmetro do render**. Enviaremos algumas informações:

```
res.render('home', {  
  'nome': 'Daniel',  
  'idade': 90  
})
```

Veremos como receber essas informações lá no view (home):

No nosso home.mst, inserimos o seguinte comando:

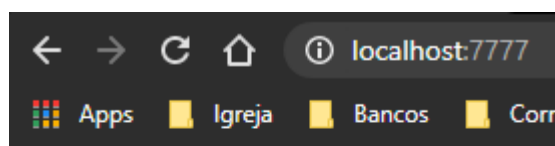
<p>Nome: {{nome}}</p> → Este comando receberá a variável de nome **'nome'** enviada pelo segundo parâmetro do render.

```
aula02 > views > home.mst  
1 <h1>Seja bem vindo</h1>  
2 <p>Qualquer mensagem</p>  
3 <p>Nome: {{nome}}</p>
```

Outra forma que podemos trabalhar lá no nosso index.js é criarmos uma variável, inserir as informações que quisermos enviar nela e no segundo parâmetro do render, declararmos apenas o nome da variável, conforme print abaixo:

```
aula02 > routes > JS index.js > router.get('/') callba  
1 const express = require('express')  
2  
3 const router = express.Router()  
4 router.get('/', (req, res)=>{  
5   let obj = {  
6     'nome': 'Daniel',  
7     'idade': 90  
8   }  
9   res.render('home', obj)  
10 }  
11  
12 module.exports = router
```

Navegador:



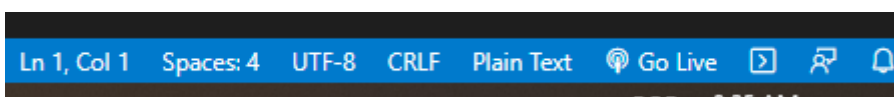
Seja bem vindo

Qualquer mensagem

Nome: Daniel - Idade: 40

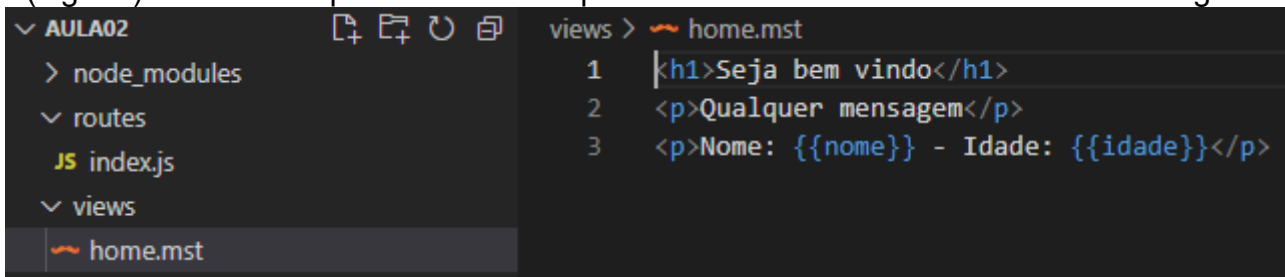
Aula 06 – Entendendo Templates (1/2)

Primeiro ponto, percebemos que os comandos do nosso arquivo home.mst estão todos brancos, e vemos no canto inferior direito da tela do VSCode a descrição Plain text. Para corrigir este “problema” iremos instalar uma extensão no VSCode.

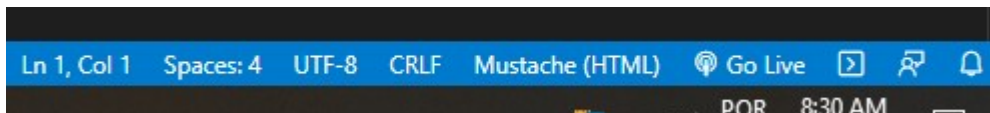


Procuraremos nas extensões do VSCode por Mustache e instalaremos a extensão que tem por nome **Mustache** que hoje (19/05/2021) está na versão 1.1.1

Com a extensão instalado, agora, o VSCode reconhece a extensão mst. Ele adiciona um ícone de mustache(bigode) no nosso arquivo home.mst e passa a colorir os comandos do nosso código:



```
views > home.mst
1 | <h1>Seja bem vindo</h1>
2 | <p>Qualquer mensagem</p>
3 | <p>Nome: {{nome}} - Idade: {{idade}}</p>
```



Com esta extensão instalada, vamos voltar a trabalhar no nosso código. Abrindo o index.js temos os seguintes comandos da aula passada:

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

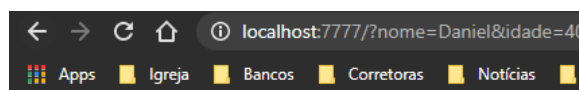
const router = express.Router()
router.get('/', (req, res)=>{
  let obj = {
    'nome': 'Daniel',
    'idade': 40
  }
  res.render('home', obj)
})

module.exports = router
```

No 'nome', ao invés de declararmos um nome específico aqui, podemos utilizar o comando **req.query.nome** para receber este valor de uma requisição. Também utilizamos para a idade

```
1 | const express = require('express')
2 |
3 | const router = express.Router()
4 | router.get('/', (req, res)=>{
5 |   let obj = {
6 |     'nome': req.query.nome,
7 |     'idade': req.query.idade
8 |   }
9 |   res.render('home', obj)
10 | })
11 |
12 | module.exports = router
```

Desta forma, podemos adicionar o nome na url (via GET neste caso) que será enviado para a home.mst via render:



Seja bem vindo

Qualquer mensagem

Nome: Daniel - Idade: 40

Obs.: A partir deste momento, o Bonieky retirou as aspas simples dos campos nome e idade (Por que? Não foi explicado, porque retirar agora ou porque utilizando antes.). Seguimos...

adicionamos um boolean (**mostrar**) ao nosso objeto obj no index.js:

Podemos (possibilidades) utilizar este boolean para fazer ele mostrar o conteúdo acima, apenas se este mostrar ficar como true;

```
router.get('/', (req, res)=>{
  let obj = {
    nome:req.query.nome,
    idade:req.query.idade,
    mostrar:false
  }
  res.render('home', obj)
})
```

E no nosso home.mst adicionamos um teste condicional com a seguinte estrutura:
{{#mostrar}}...execução...{{/mostrar}}

```
views > ~ home.mst
1 <h1>Seja bem vindo</h1>
2 <p>Qualquer mensagem</p>
3 {{#mostrar}}
4 <p>Nome: {{nome}} - Idade: {{idade}}</p>
5 {{/mostrar}}
```

Obs.: Aqui, foi constatado que 'mostrar' é um boolean por isto ele realiza o teste lógico.

Podemos também criar loops: Veja a estrutura:
index.js:

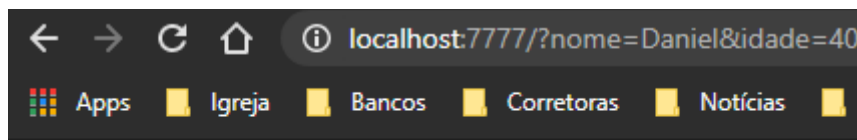
```
router.get('/', (req, res)=>{
  let obj = {
    nome:req.query.nome,
    idade:req.query.idade,
    mostrar:true,
    ingredientes:[
      {nome:'Arroz', qt:'20g'},
      {nome:'Macarrão', qt:'100g'}
    ]
  }
  res.render('home', obj)
})
```

Aqui, criamos um array (ingredientes) e adicionamos objetos dentro dele.
No home.mst adicionamos a seguinte estrutura:

```
<ul>
  {{#ingredientes}}
    <li>{{nome}} {{qt}}</li>
  {{/ingredientes}}
</ul>
```

Obs.: Aqui, foi constatado que 'ingredientes' é um array por isso ele pode criar um contador.

Navegador:



Seja bem vindo

Qualquer mensagem

Nome: Daniel - Idade: 40

- Arroz - QT: 20g
- Macarrão - QT: 100g

Aula 07 – Entendendo Templates (2/2)

Adicionamos o seguinte array no nosso objeto obj:

```
interesses: ['node', 'js', 'css']
```

Aqui criamos um array sem objetos dentro. Veja no home.mst como fazemos o contador para o array simples (Sem objetos)

```
<ul>
  {{#interesses}}
  <li>{{.}}</li>
  {{/interesses}}
</ul>
```

Este ponto dentro das chaves, faz com que o contador passe por todos os itens deste array.

Como **mostrar ou não mostrar uma tag html** no navegador. Veja o exemplo.

Criamos o seguinte elemento no nosso objeto:

```
teste: '<strong>Testando negrito</strong>'
```

Se no home .mst apenas chamarmos o elemento teste, ele irá apresentar na tela toda a informação contida nele, inclusive as tags.

```
{{teste}}
```

Navegador:

Testando negrito

Para que as tags não apareçam, apenas adicionamos no {{teste}} do home, mais um par de chaves:

```
{{{teste}}}
```

Navegador:

Testando negrito

Vejamos outros detalhes:

Voltando a ver o nosso **array ingredientes**:

Vamos supor que não há ingredientes neste array, como fazemos para informar que não há ingredientes:

Para isto, utilizamos a seguinte estrutura no home.mst:

```
<ul>
  {{#ingredientes}}
    <li>{{nome}} {{qt}}</li>
  {{/ingredientes}}
</ul>
{{^ingredientes}}
<p>Não há ingredientes</p>
{{/ingredientes}}
```

Observe que aqui, abaixo dos comando que listamos itens deste array adicionamos a estrutura com o nome do array antecedido do símbolo ^. Após essa abertura, adicionamos um parágrafo com um aviso que apareça na tela, caso este array esteja vazio e depois fechamos este trecho com o / → {{/ingredientes}}

Para testarmos estes comandos, podemos ir no nosso index.js e esvaziar este array. Fazendo isto o alerta é exibido no navegador:

Não há ingredientes

Podemos também adicionar comentários no código do arquivo .mst das seguintes forma:

```
{{! Comentários}}
<!--Comentários-->
```

Embora, esta forma igual aos comentários do html, eu acredito (eu acho) que só funcione aqui, devido a extensão mustache que instalamos no VSCode.

Até aqui, o nosso home.mst ficou assim:

```
views > home.mst
1  <h1>Seja bem vindo</h1>
2  <p>Qualquer mensagem</p>
3  {{#mostrar}}
4  <p>Nome: {{nome}} - Idade: {{idade}}</p>
5  {{/mostrar}}
6  <ul>
7  {{#ingredientes}}
8  |   <li>{{nome}} {{qt}}</li>
9  {{/ingredientes}}
10 </ul>
11 {{^ingredientes}}
12 <p>Não há ingredientes</p>
13 {{/ingredientes}}
14 <hr>
15 <ul>
16 {{#interesses}}
17 <li>{{.}}</li>
18 {{/interesses}}
19 </ul>
20 <hr>
21 {{teste}}
22 {{teste}}
```

A partir daqui, iremos apagar as informações deste arquivo para fazermos novos testes:

Uma prática muito utilizada na criação de um site, é manter as características visuais do meu site nas diversas páginas que ele contém. Levando isto em consideração, uma prática muito comum, é criarmos um arquivo padrão para o site, e cada página dele busque estas informações deste arquivo. Desta forma,

sempre que houver uma modificação neste arquivo, esta modificação será replicada automaticamente a todas as páginas deste site.

Um exemplo muito comum desta aplicação é a replicação do cabeçalho do site:

Esta prática chama-se **partials** no Node.

Para isto voltaremos a editar o nosso o arquivo **app.js**

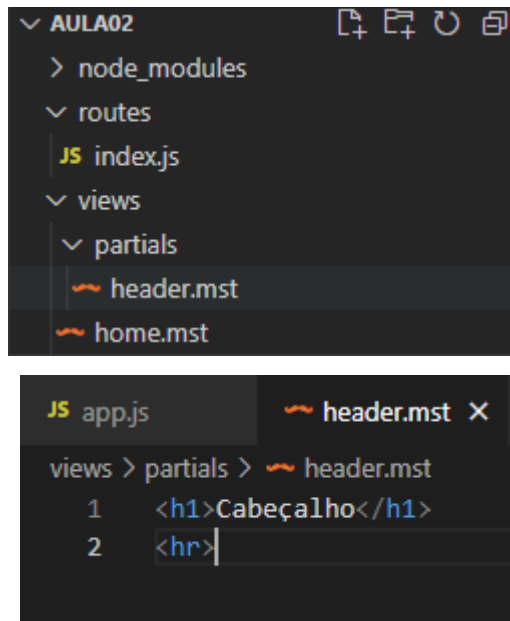
Agora é o momento que começaremos a utilizar os parâmetros aqui na função mustache.

```
app.engine('mst', mustache(__dirname+'/views/partials'))
```

O **primeiro parâmetro** serve para indicar a pasta que estão armazenados esses partials separados. Aqui, apontamos para uma pasta chamada **partials** que está dentro da pasta **views**. Como esta pasta ainda não existe, criaremos ela agora.

Obs.: na indicação do diretório, é essencial iniciarmos a declaração após o `__dirname+` com o `/`, se não, o endereço não conseguirá encontrar o destino. (Experiência própria :))

E, já para começarmos a utilizar estas partials, vamos criar, dentro desta pasta um arquivo chamado de **header.mst** e já adicionarmos um `<h1>` nele.



O **segundo parâmetro** serve para indicar qual é a extensão dos arquivos partials.

```
app.engine('mst', mustache(__dirname+'/views/partials', '.mst'))
```

A partir daqui, podemos chamar os partials existentes no `home.mst` da seguinte maneira:

```
{{>header}}
```

E ele funcionará normalmente.

Terminamos esta aula com os seguintes arquivos da seguinte forma:

index.js: Não houve modificações:

```

routes > JS index.js > router.get('/') callback > obj > teste
1  const express = require('express')
2
3  const router = express.Router()
4  router.get('/', (req, res)=>{
5    let obj = {
6      nome:req.query.nome,
7      idade:req.query.idade,
8      mostrar:true,
9      ingredientes:[
10       {nome:'Arroz', qt:'20g'},
11       {nome:'Macarrão', qt:'100g'}
12     ],
13     interesses:['node','js','css'],
14     teste:'<strong>Testando negrito</strong>'
15   }
16   res.render('home', obj)
17 })
18
19 module.exports = router

```

app.js:

```

JS app.js > ...
1  const express = require('express')
2  const mustache = require('mustache-express')
3  const router = require('./routes/index')
4
5  //Configurações
6  const app = express()
7  app.use('/', router)
8
9  app.use(express.json())
10
11 app.engine('mst', mustache({__dirname+'/views/partials','.mst'}))
12 app.set('view engine', 'mst')
13 app.set('views', __dirname + '/views')
14
15 module.exports = app

```

header.mst:

```

views > partials > header.mst
1  <h1>Cabeçalho</h1>
2  <hr>

```

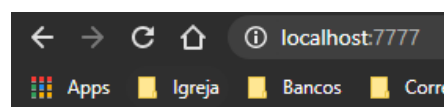
home.mst:

```

views > home.mst
1  {{>header}}
2
3  <h1>Página Home</h1>
4

```

Navegador:



Cabeçalho

Página Home

Aula 08 – Templates Helpers

Teplates Helpers → Configuração que vamos fazer na nossa aplicação que vai permitir termos vários dados (termos acesso a vários dados) em toda a nossa aplicação independente

Para esta aula, estamos esvaziando os conteúdos da variável `obj` do nosso `index.js`. Iniciando a aula com o nosso **index.js** assim:

```
routes > JS index.js > router.get('/') callback > obj
1  const express = require('express')
2
3  const router = express.Router()
4  router.get('/', (req, res)=>{
5    let obj = {}
6
7
8    res.render('home', obj)
9  })
10
11 module.exports = router
```

Adicionado o item `pageTitle: 'Título de Teste'`

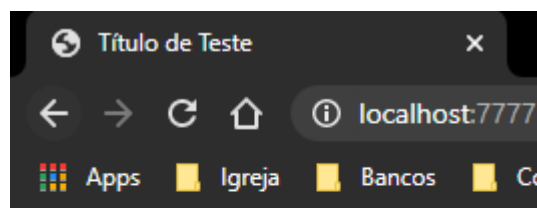
```
let obj = {
  pageTitle: 'Título de Teste'
}
```

No arquivo **home.mst** alteramos o `h1` para `<h1>Página Home</h1>`

E no **header.mst** criamos a seguinte estrutura:

```
JS app.js  X  header.mst  X  JS index.js
views > partials > header.mst
1  <html>
2  <head>
3    <title>{{pageTitle}}</title>
4  </head>
5  <body>
6    <h1>Cabeçalho</h1>
7    <hr>
8  </body>
```

Navegador:



Cabeçalho

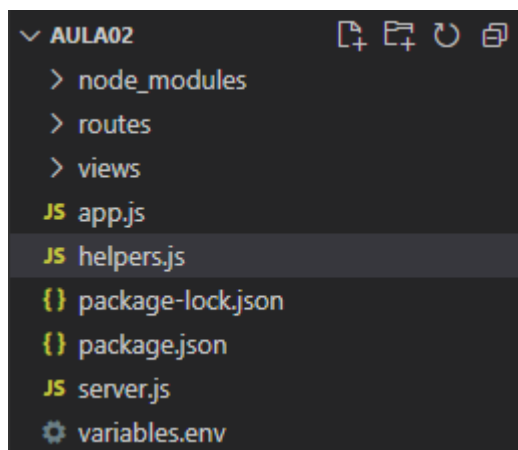
Página Home

Observe que no título da página, ficamos com o conteúdo de pageTitle.

Agora veremos a utilização dos Helpers

Adicionamos informações padrão para todos os views, para que em caso, alguma informação não seja adicionada, o helper possa enviar esta informação padrão. Neste exemplo acima, seria o caso, se não houvesse sido inserido um título na página, o helper entraria em ação com um título padrão.

Para isto, criaremos um arquivo na raiz do projeto e iremos chamá-lo de **helpers.js** (De acordo com o Bonieky, este é um nome que convencionalmente é utilizado para este tipo de aplicação).



Editando o helpers.js:

exports.defaultPageTitle = "Site ABC" → ativa a exportação do conteúdo que chamamos de defaultPageTitle.

Novamente, limpamos o conteúdo da nossa variável obj do **index.js**.

E no **app.js** realizamos as seguintes modificações:

const helpers = require('./helpers') → Criamos a variável helpers e importamos para o app.js seu conteúdo

Este comando abaixo, precisa ser criado **OBRIGATORIAMENTE antes do 'app.use('/', router)'** para que ele funcione.

```
app.use((req, res, next)=>{
  res.locals.h = helpers
  res.locals.teste = "123"
  next()
})
```

Aqui, esta função, possui três parâmetros. Porém, inicialmente utilizaremos o parâmetro 'res'.

locals → Faz com que as nossas variáveis se tornem globais para a nossa aplicação.

res.locals.h = helpers → **h** → Tem gente que utiliza apenas o 'h' (de helpers) mas este 'h' aqui é apenas o nome da variável em que estamos armazenando o helpers.

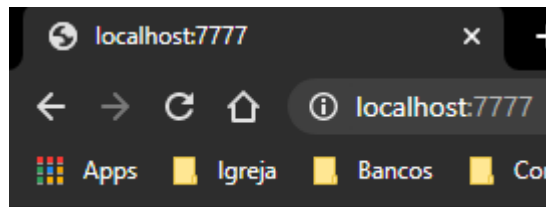
res.locals.teste = "123" → Criamos esta variável para realizarmos testes.

Next() → **next** → Pega as informações e envia para a próxima página que for acessada.

E no home.mst, fizemos a seguinte alteração:

```
<h1>Página Home: {{teste}}</h1>
```

Navegador:



Cabeçalho

Página Home: 123

Agora, aqui para prosseguirmos com os testes, realizamos as seguintes modificações:

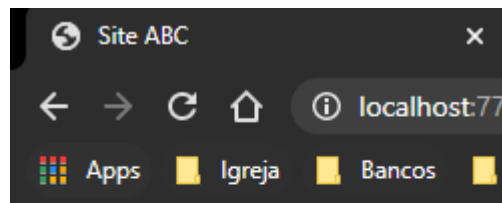
- Removemos a linha : **res.locals.teste = "123"** do app.js.
- Removemos **{{teste}}** do home.mst

No **head** do nosso **header.mst**, adicionamos o seguinte:

```
<head>
  {{#pageTitle}}
  <title>{{pageTitle}}</title>
  {{/pageTitle}}
  {{^pageTitle}}
  <title>{{h.defaultPageTitle}}</title>
  {{/pageTitle}}
</head>
```

Para executar o pageTitle se houver, e se não houver, adicionar o defaultPageTitle

Navegador:



Cabeçalho

Página Home

Pela falta de pageTitle, foi adicionado a informação do helper 'Site ABC'

Para a criação de um menu, utilizamos a seguinte estrutura, para evitarmos retrabalho.

Aqui, adicionamos os comandos a seguir, dentro do helpers.js, mas poderíamos criar um novo arquivo específico para os menus

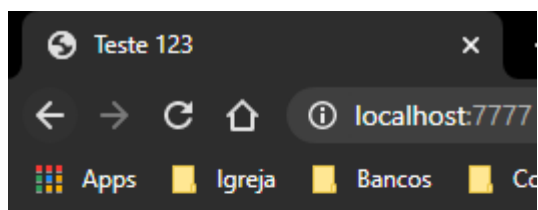
No helpers.js adicionamos o export menu: contendo um array com os seguintes objetos

```
JS app.js JS helpers.js X header.mst
JS helpers.js > [0] menu > slug
1 exports.defaultPageTitle = "Site ABC"
2 exports.menu = [
3   {name: 'Home', slug: '/'},
4   {name: 'Sobre', slug: '/sobre'},
5   {name: 'Contato', slug: '/contato'}
6 ]
```

E no header.mst, adicionamos um contador para montar o menu:

```
<ul>
  {{#h.menu}}
  <li><a href="{{{slug}}}">{{{name}}}</a></li>
  {{/h.menu}}
</ul>
```

Navegador:



Cabeçalho

- [Home](#)
- [Sobre](#)
- [Contato](#)

Página Home

Obs Geral:

- Sobre o helper do <title>, uma outra opção apresentada:

Ao invés de:

```
<head>
  {{#pageTitle}}
  <title>{{{pageTitle}}}</title>
  {{/pageTitle}}
  {{{^pageTitle}}
  <title>{{{h.defaultPageTitle}}}</title>
  {{/pageTitle}}
</head>
```

Podemos utilizar

```
<head>
  <title>{{{pageTitle}} }{{{h.defaultPageTitle}}}</title>
</head>
```

Porém, neste segundo exemplo, se houver o **pageTitle** e o **h.defaultPageTitle** os dois aparecerão no título, porém, se não for adicionado nada no **pageTitle**, aparecerá o **h.defaultPageTitle** normalmente, não causando nenhuma falha visual.

Resumindo, criamos o nosso **helpers.js**, exportamos ele para a variável **h** no nosso **app.js** e utilizamos no nosso **header.mst**.

helpers.js:

```
JS app.js  X  JS helpers.js  X  header.mst
JS helpers.js > menu > slug
1  exports.defaultPageTitle = "Site ABC"
2  exports.menu = [
3    {name: 'Home', slug: '/'},
4    {name: 'Sobre', slug: '/sobre'},
5    {name: 'Contato', slug: '/contato'},
6    {name: 'Teste', slug: '/teste'}
7  ]
```

app.js:

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

header.mst:

```
JS app.js  JS helpers.js  header.mst X  JS index.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 </body>
```

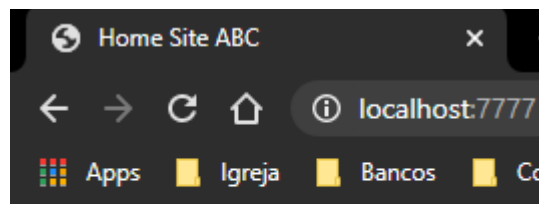
index.js:

```
JS app.js  JS helpers.js  header.mst  JS index.js  X
routes > JS index.js > [?] <unknown>
1  const express = require('express')
2
3  const router = express.Router()
4  router.get('/', (req, res)=>{
5    let obj = {
6      pageTitle:"Home"
7    }
8  }
9  res.render('home', obj)
10 })
11
12 module.exports = router
```

home.mst:

```
JS app.js  JS helpers.js  header.mst  JS index.js  home.mst X
views > home.mst
1  {{>header}}
2
3  <h1>Página Home</h1>
4  |
```

Navegador:



Cabeçalho

- [Home](#)
- [Sobre](#)
- [Contato](#)
- [Teste](#)

Página Home

Aula 09 – Estrutura MVC no Node

Padrão de arquitetura MVC:

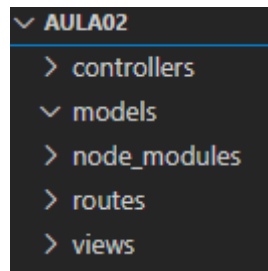
M → **Model** → Responsável pela organização de dados (organização das informações), processamento de dados.

V → **View** → Responsável pela parte Visual (mustache)

C → **Controller(controlador)** → É responsável por organizar toda a informação de modo geral ele linka o model ao view

MVC é uma forma de separar os arquivos por área de responsabilidade do sistema. Para organizar esta estrutura, utilizaremos três pastas, uma para cada uma destas funções.

A pasta views, nós já temos, portanto, criaremos agora as pastas models e controllers, na raiz do nosso projeto:



controllers → É criado um arquivo controller para cada parte específica do nosso sistema. Iniciaremos criando o arquivo **homeController.js** dentro da pasta controllers.

Agora, transportaremos as funções de rotas do arquivo **index.js** para a seção de **controllers**, pois é lá que esta função deve ficar.

Retiramos esta seção abaixo, exatamente o que está selecionado, do index.js:

```
const router = express.Router()
router.get('/', (req, res) => {
  ... let obj = {
  ...   pageTitle: "Home"
  ... }
  ... res.render('home', obj)
})
```

E no nosso arquivo **homeController.js** criamos o um **export** com o nome de index (para o principal(home)) e adicionamos a nossa rota para a raiz do sistema:

```
exports.index = (req, res) => {
  let obj = {
    pageTitle: "Home"
  }
  res.render('home', obj)
})
```

Já que possibilitamos a exportação da rota principal no homeControllers, agora, no index.js, precisamos importar (require) esta rota. Para isto adicionamos os seguintes comandos:

a variável:

const homeController = require('../controllers/homeController')

e o parâmetro:

homeController.index

no **router**

index.js fica assim:

```
routes > JS index.js
1  const express = require('express')
2  const homeController = require('../controllers/homeController')
3
4  const router = express.Router()
5  router.get('/', homeController.index)
6
7  module.exports = router
```

A partir daqui, iremos criar um outro controller e uma outra página, para exercitar. Para isto, vamos acessar o arquivo helpers.js e criar um menu de login. Para isto adicionamos a seguinte linha:

```
{name:'Login',slug:'/users/login'}
```

Voltamos para o nosso index.js para criarmos a rota 'users/login:

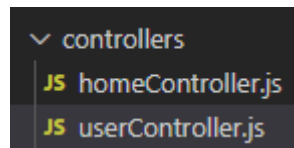
Para isto, criamos a variável userController:

```
const usersController = require('../controllers/userController')
```

E criamos também a sua rota:

```
router.get('/users/login', userController.login)
```

Obs.: Embora, tenhamos criado aqui a variável e a rota para userController, ainda não criamos este arquivo. Apenas agora iremos criá-lo lá na pasta controllers:

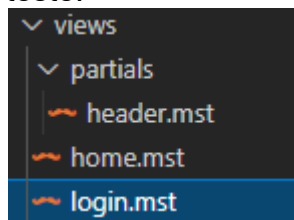


E adicionamos o exports no userController:

```
controllers > JS userController.js
1 exports.login = (req, res) => {
2   res.render('login')
3 }
```

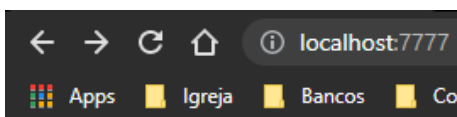
Agora, para que a página login, possa ser efetivamente renderizada, precisamos criá-la. Dentro da pasta views, criamos o arquivo login.mst.

Criamos um arquivo simples, apenas para teste:



```
views > login.mst
1 <h1>Faça o Seu Login</h1>
2 <form>
3   <input type="text" name="usuario">
4   <button>Entrar</button>
5 </form>
```

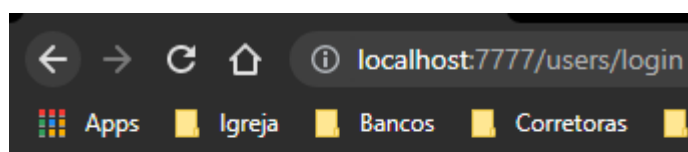
HTML:



Cabeçalho

- [Home](#)
- [Sobre](#)
- [Contato](#)
- [Teste](#)
- [Login](#)

Página Home



Faça o Seu Login

No final da aula, criamos uma rota para registro de usuários, apenas para visualizarmos a facilidade que este sistema de MVC trás para o desenvolvimento do sistema.

Index.js:

```
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
```

userController.js:

```
controllers > JS userController.js > login > exports.login
1  exports.login = (req, res) => {}
2    res.render('login')
3  }
4  exports.register = (req, res) => {
5    res.render('register')
6  }
```

Aula 10 – Middleware

Middleware → Código que será inserido no meio (middleware) de um processo.

Exemplo: Processo de login:

Quando um usuário digita seu e-mail e sua senha para acessar um sistema, ele está fazendo uma requisição.

Para esta requisição, o sistema deve retornar uma resposta, que pode ser positiva ou negativa.

Neste processo, tudo o que acontece no meio, entre a requisição e a resposta, no Node, geralmente é feito por middlewares. Exemplos: Validar campos, autorizar usuário.

Existem 2 tipos de middleware no Node:

- Global → Para toda a aplicação. Exemplo: os **use** que criamos no app.js
- Local → Específico em uma aplicação.

Primeira interferência no código:

No app.js, trocamos o **app.use(express.json())** de lugar com o **app.use('/', router)** estes são middleware globais que já estávamos utilizando e de acordo com a orientação do Bonieky, o ideal é que o **router** fique por último entre os middleware. Assim:

```
JS app.js X
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
6  //Configurações
7  const app = express()
8  app.use((req, res, next)=>{
9    res.locals.h = helpers
10   next()
11 })
12 app.use(express.json())
13 app.use('/', router)
14
15 app.engine('mst', mustache(__dirname+'/views/partials','.mst'))
16 app.set('view engine', 'mst')
17 app.set('views', __dirname + '/views')
18
19 module.exports = app
```

Vamos agora criar um middleware local:
Este middleware irá adicionar informações em uma requisição de login.

Vamos criar no homeController.js uma nova função:

```
controllers > JS homeController.js > index > exports.index > obj
1  exports.userMiddleware = (req, res, next) => {
2    let info = {name:'Daniel', id:123}
3    req.userInfo = info
4    next()
5  }
6  exports.index = (req, res)=>{
7    let obj = {}
8    pageTitle:"Home",
9    userInfo: req.userInfo
10   }
11   res.render('home', obj)
12 }
```

exports.userMiddleware = (req, res, next) => { → userMiddleware → Nome sugestivo
let info = {name:'Daniel', id:123} → Estamos utilizando esta linha para simular a coleta destas informações como se fosse de um banco de dados.
req.userInfo = info → Adicionamos a informação na requisição (userInfo -> nome sugerido)
next() → Adicionamos o next() aqui para que após a execução do userMiddleware, a próxima tarefa seja executada.
userInfo: req.userInfo → Criamos este item com o mesmo nome(userInfo e passamos todas as informações da requisição)

home.mst:

```
views > home.mst
1  {{>header}}
2  <h1>Página Home</h1>
3  <p>USUÁRIO LOGADO: {{userInfo.name}}</p>
```

<p>USUÁRIO LOGADO: {{userInfo.name}}</p> Adicionamos esta linha no home, mas ainda não é suficiente para que as informações apareçam na tela. Para isto, precisamos adicionar o userMiddleware na rota da página home (raiz, /) no nosso arquivo index.js.

Index.js:

```
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.userMiddleware, homeController.index)
7  router.get('/users/login', userController.login)
8  router.get('/users/register', userController.register)
9
10 module.exports = router
```

router.get('/', homeController.userMiddleware, homeController.index) → Aqui, adicionamos o homeController.userMiddleware no segundo parâmetro para usarmos o middleware que criamos na página home.