**NodeJS – Módulo 02**

**Material de estudo desenvolvido por:**

**Daniel Teixeira Quadros**

**Sumário**

[Aula 01 – Variáveis de Projeto 2](#__RefHeading___Toc1030_957117295)

[Aula 02 – Configurando o MongoDB (Banco de Dados) 4](#__RefHeading___Toc1032_957117295)

[Aula 03 – Entendendo as Rotas (1/2) 8](#__RefHeading___Toc1034_957117295)

[Aula 04 – Entendendo as Rotas (2/2) 10](#__RefHeading___Toc1036_957117295)

[Aula 05 – Configurando Template Engine 12](#__RefHeading___Toc1038_957117295)

[Aula 06 – Entendendo Templates (1/2) 14](#__RefHeading___Toc1040_957117295)

[Aula 07 – Entendendo Templates (2/2) 17](#__RefHeading___Toc1042_957117295)

[Aula 08 – Templates Helpers 21](#__RefHeading___Toc1044_957117295)

[Aula 09 – Estrutura MVC no Node 26](#__RefHeading___Toc1046_957117295)

[Aula 10 – Middleware 29](#__RefHeading___Toc1048_957117295)

# 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=developmente** → 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.

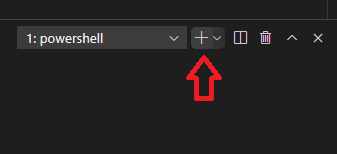
**PORT=7777** → Porta, utilizaremos a padrão ‘7777’, aí se quisermos mudar a porta, é só alterarmos este valor e o servidor mudará a porta.

**DATABASE=mongodb://127.0.0.1:27017** → Configurações para conexão com o Banco de Dados.

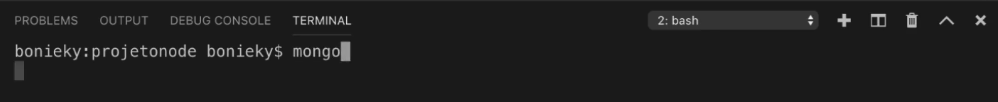
Como saber esta configuração? A Configuração do servidor local é muito simples de saber. Para isto utilizaremos o terminal:

**Obs, se tivesse usuário e senha, utilizariamos aqui ‘at @ senha’.**

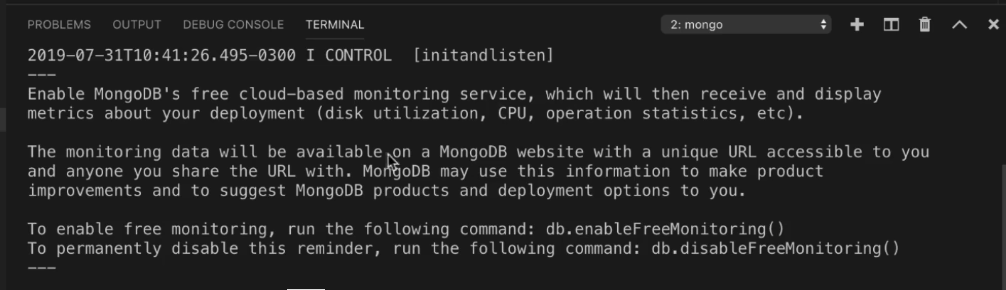
Obs.: Não utilizaremos o que está rodando, mas abriremos um novo:



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:



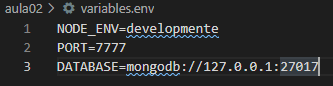
E no meu deu erro de comando desconhecido.

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, copiei 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.

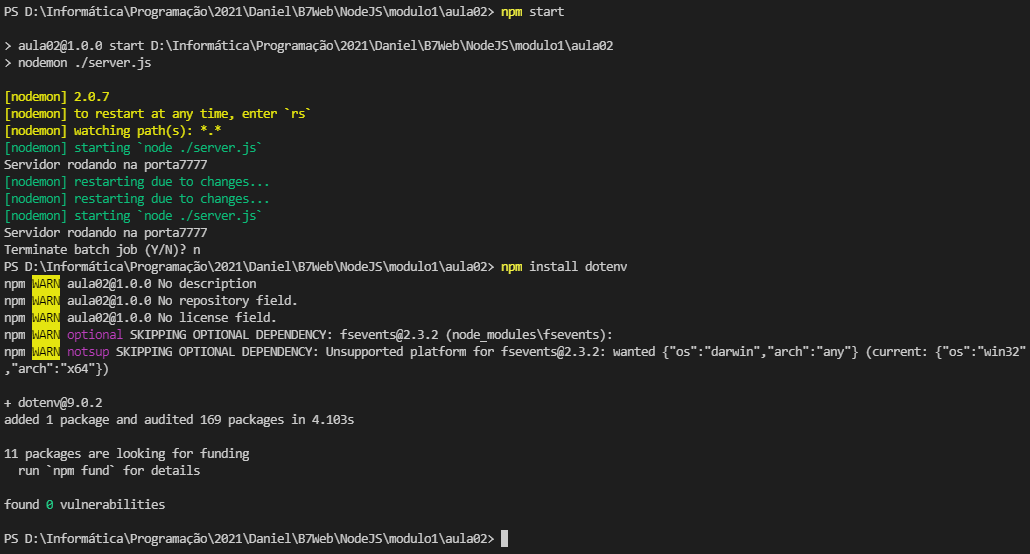
Por enquanto, o nosso arquivo variables.env ficou assim:



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 extenção chamada **dotenv** que é utilizada para isso.

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

Obtivemos o seguinte resultado:



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 extenção **.env** a vairável de nome PORT (Aquela que criamos lá no arquivo 'variables.env') e, por segurança adicionamos o (**|| 7777**). Que, se por acaso, não encontrar a variável PORT, uitilizará a porta '7777'.

Terminamos a aula com o server.js assim:

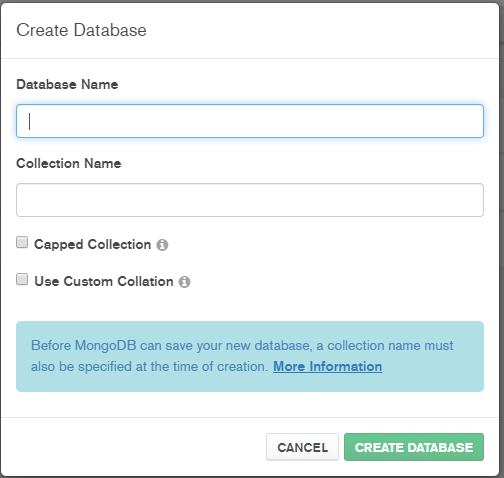


**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# 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:



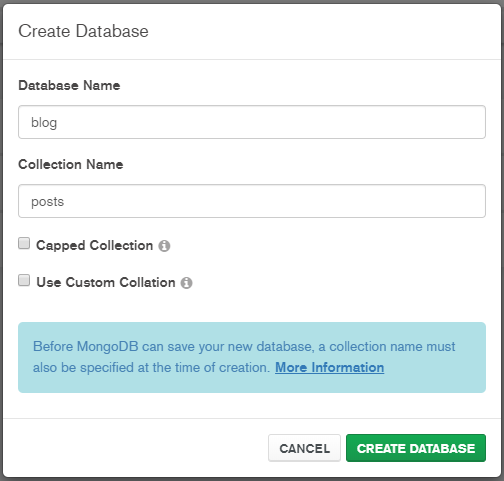
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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 →

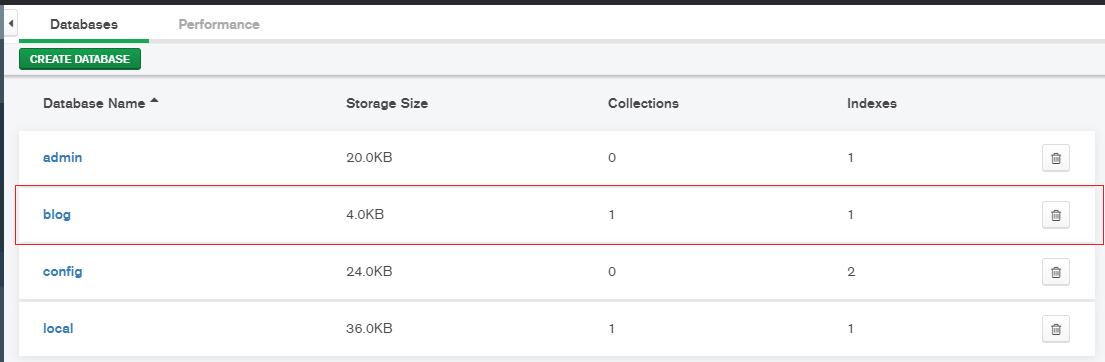
Collection Name →

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

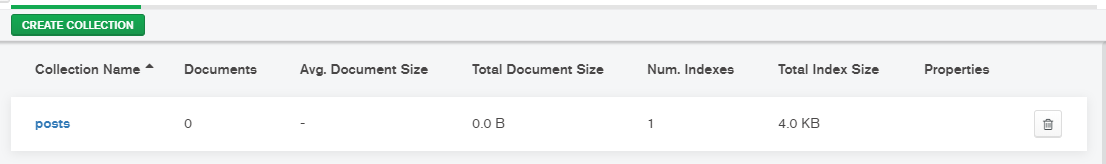


\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

O nosso Banco de Dados já aparece aqui na lista:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

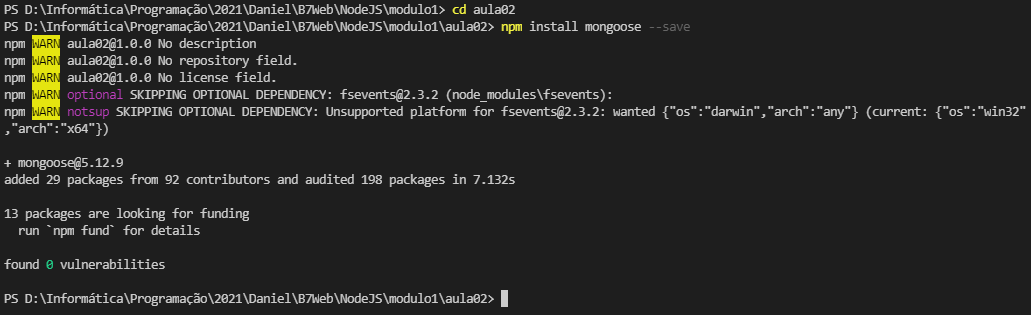
Clicando no nosso Banco de Dados, já aparece a nossa collection ‘posts’:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

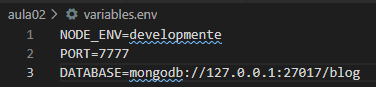
Para fazermos a conexão do Banco de Dados com o nosso sistema, utilizaremos uma dependência chamado ‘mongoose’. Para isto, executamos o seguinte comando no terminal:

**npm install mongoose --save**



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Após adicionarmos a nossa colection, 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:

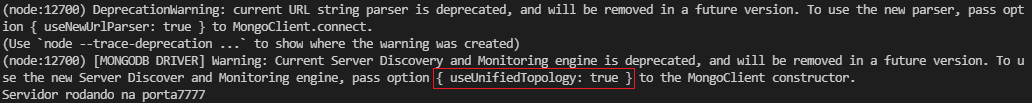
1º - Informamos o que estamos monitorando. Neste caso, são os erros ('**error**').

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.

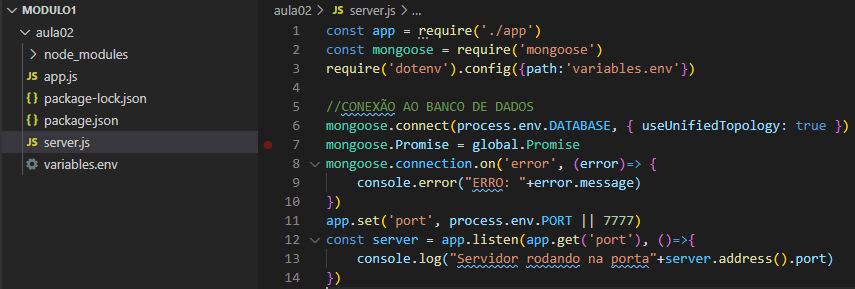
Neste caso, vamos acatar ao aviso que está dizendo que o **url parser** que é o padrão do connect, ele vai mudar e vai ser removido:



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 })



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# 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 administrção. 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 utilizariamos 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)=>{

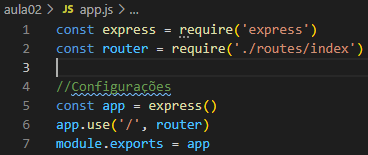
    res.send('Olá Mundo!')

})

Adicionamos o seguinte comando:

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

**O nosso app.js ficou assim:**



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

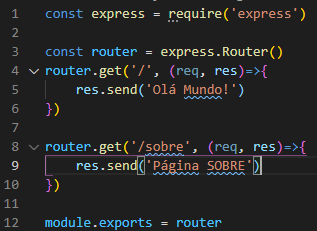
router.get('/sobre', (req, res)=>{

    res.send('Página SOBRE')

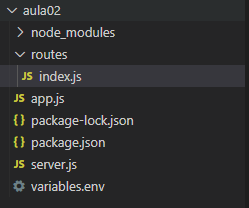
})

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

Nosso arquivo index.js ficou assim:



E a estrutura no VSCode, ficou assim:



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# 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)

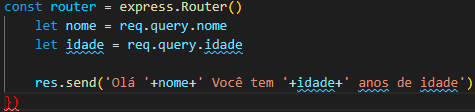
**Como acessamos esses conteúdos?**

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

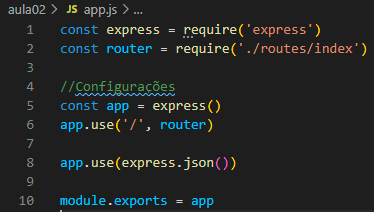
**http://localhost:7777/?nome=Daniel&idade=40**, neste exemplo, embora não tenha ocorrido nenhuma alteração em nossa página ou site, eles estão disponívei no 'req'

E para acessá-los

Aqui utilizamos o send. Para visualizarmos:



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



**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:

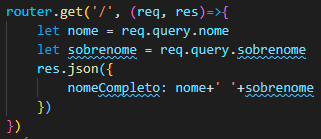
**router.get('/posts/:id', (req, res)=>{ →**  Aqui, o '**posts**' dá a ideia da rota dos posts e o **:id** é utilizado para pegarmos um post com um id específico que será vinculado a ele.

**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:**



**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.



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

GET: req.query

POST: req.body

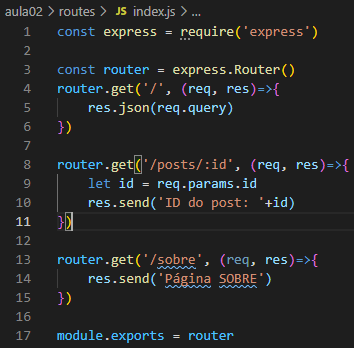
PARÂMETROS DA URL: req.params

**ENVIO:**

SEND

JSON

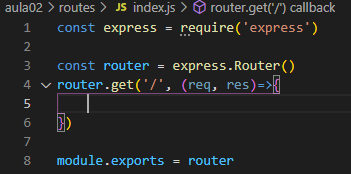
Nosso index.js ficou assim:



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

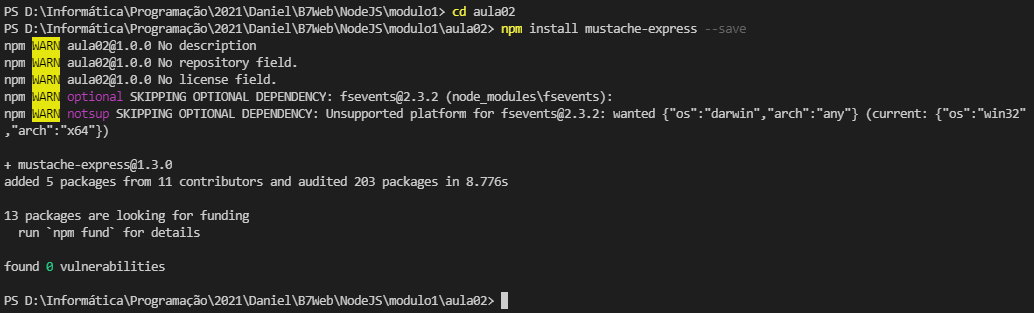
# Aula 05 – Configurando Template Engine

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



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:



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

Trazemos a pliacação 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 párâmetro.

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.

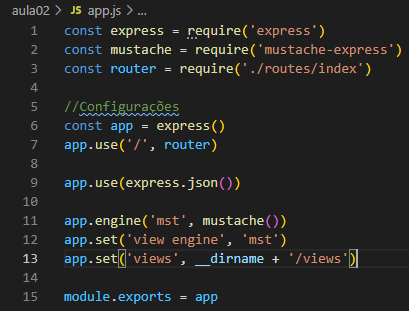
Para isto, criaremos uma pasta que chamamos de ‘**views**’ na raiz do projeto

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:



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

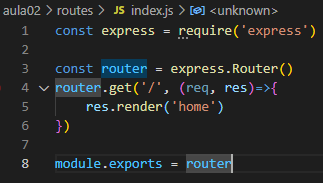
na rota raiz da nossa aplicação, ao invés de utilizarmos um **.send** ou um **.json** utilizaremos o **.render** de renderizar:

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’.

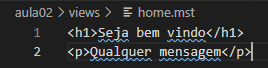
2º – Vão os dados que enviaremos para o view, mas por enquanto, não iremos utilizar.



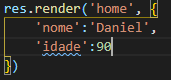
Agora, quando acessarmos a ‘**/**’ no servidor, ele vai executar o.**render(‘home’)**.

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:



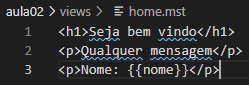
Agora, utilizaremos o **segundo parâmetro do render**. Enviaremos algumas informações:



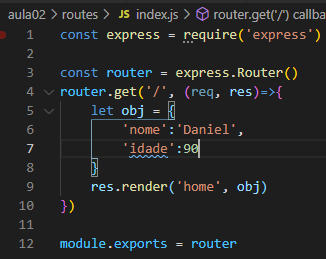
Veremos como receber essas informações lá no view (home):

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.



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:



Navegador:



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

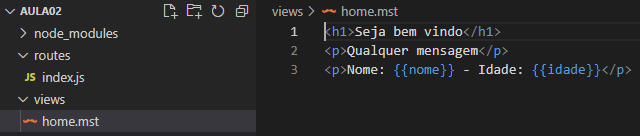
# Aula 06 – Entendendo Templates (1/2)

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.



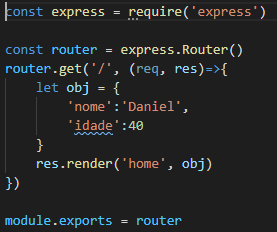
Procuraremos nas extenções do VSCode por Mustache e intalaremos a extensão que tem por nome **Mustache** que hoje (19/05/2021) está na versão 1.1.1

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:

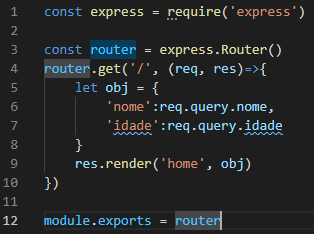




Com esta extensão instalada, vamos voltar a trabalhar no nosso código. Abrindo o index.js temos os seguintes comandos da aula passada:



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

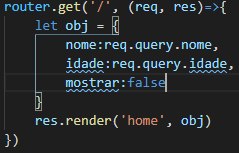


Desta forma, podemos adicionar o nome na url (via GET neste caso) que será enviado para a home.mst via render:

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 utilizardo antes.). Seguimos…

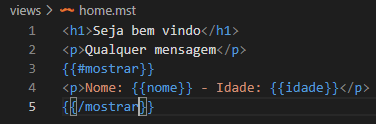
adicionamos um boolean (**mostrar**)ao nosso objeto obj no index.js:

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



E no nosso home.mst adicionamos um teste condicional com a seguinte extrutura:

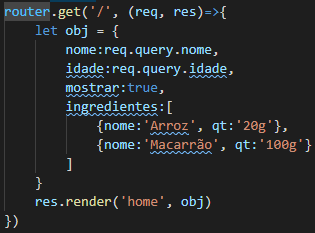
**{{#mostrar}}...execução...{{/mostrar}}**



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

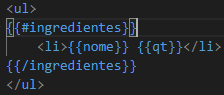
Podemos também criar loopes: Veja a estrutura:

index.js:



Aqui, criamos um array (ingredientes) e adicionamos objetos dentro dele.

No home.mst adicionamos a seguinte estrutura:



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

Navegador:



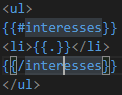
**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# 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)



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:



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

{{{teste}}}

Navegador:

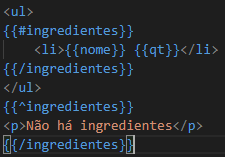


Vejamos outros detalhes:

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:



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:



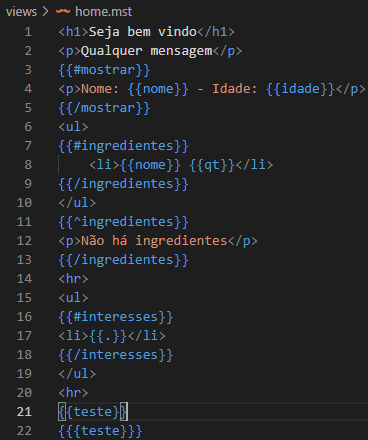
Podemos também adicionar comentários no código do arquivo .mst das seguintes forma:

{{! Comentários}}

<!--Comentários-->

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

**Até aqui, o nosso home.mst ficou assim:**

****

**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.

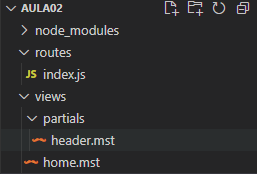
app.engine('mst', mustache(\_\_dirname+'/views/partials'))

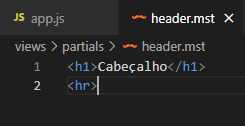
**O primeiro parâmetro** serve para indicar a pasta que estão armazenados esses partials separados.

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'))

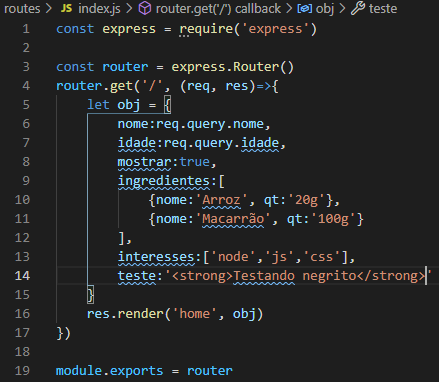
A partir daqui, podemos chamar os partials existentes no home.mst da seguinte maneira:

{{>header}}

E ele funcionará normalmente.

Terminamos esta aula com os seguintes arquivos da seguinte forma:

**index.js: Não houve modificações:**



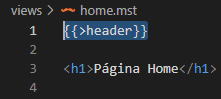
**app.js:**



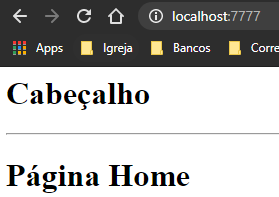
**header.mst:**



**home.mst:**



**Navegador:**

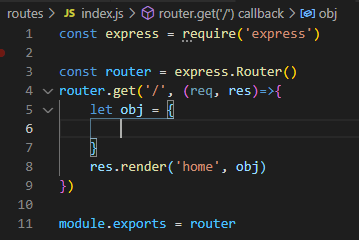
****

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

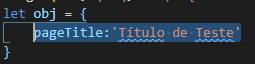
# Aula 08 – Templates Helpers

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:

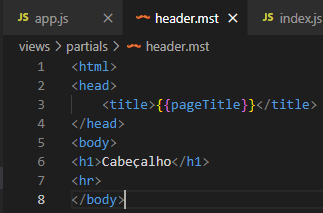


Adicionado o item pageTitle:'Título de Teste'

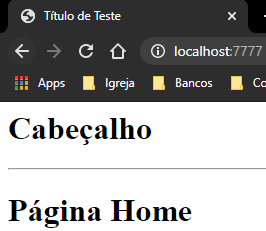


No arquivo **home.mst** alteramos o h1 para <h1>Página Home</h1>

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



Navegador:

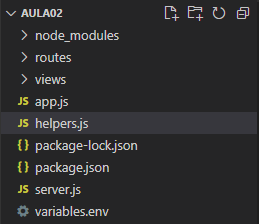


Observe que no título da página, ficamos com o conteúdo de pageTitle.

**Agora veremos a utilizada dos Helpers**

Adicionamos informações padrões 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:

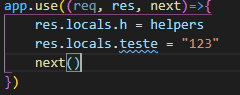
**exports.defaultPageTitle = "Site ABC"** → ativa a exportação do conteúdo que chamamos de defaultPageTitle.

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.



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.

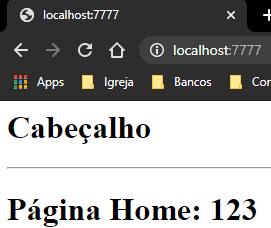
**res.locals.teste = "123" →** Criamos esta variável para realizarmos testes.

**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:

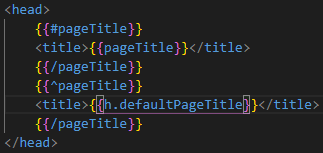


Agora, aqui para prosseguirmos com os testes, realizamos as seguintes modificações:

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

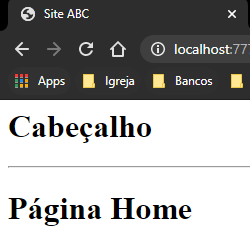
- Removemos **{{teste}}** do home.mst

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



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

Navegador:



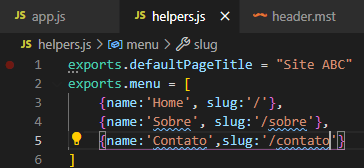
Pela falta de pageTitle, foi adicionado a informação do helper ‘Site ABC

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

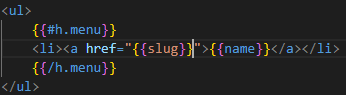
Para a criação de um menu, utilizamos a seguinte estrutura, para evitarmos retrabalho.

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

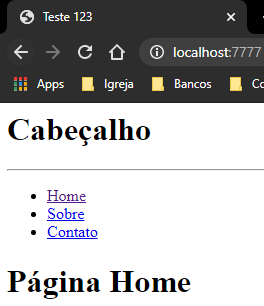
No helpers.js adicionamos o export menu: contendo um array com os seguintes objetos



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



Navegador:

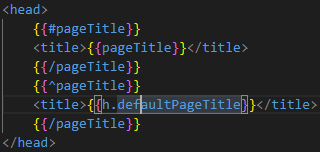


\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Obs Geral:

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

Ao invés de:



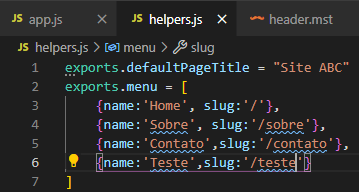
Podemos utilizar



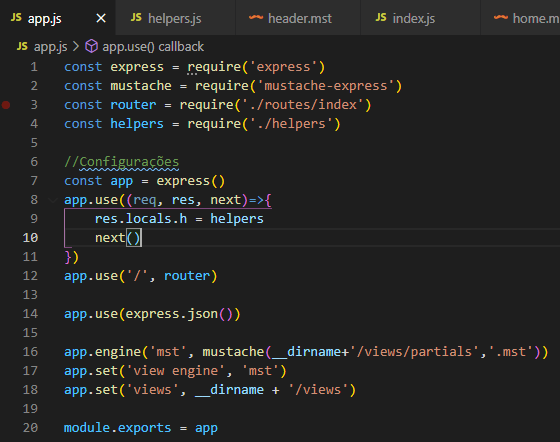
Porém, neste segundo exemplo, se houver o **pageTitle** e o **h.defaultPageTitle** os dois aparecerão no título, porém, se não for adicionado nada no **pageTitle**, aparecerá o **h.defaultPageTitle** normalmente, não causando nenhuma falha visual.

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

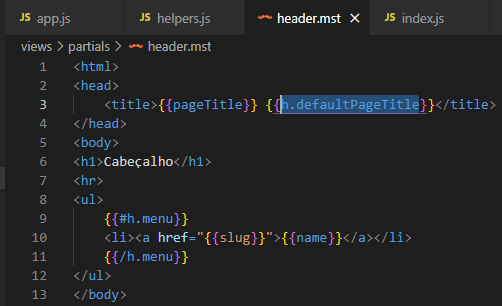
helpers.js:



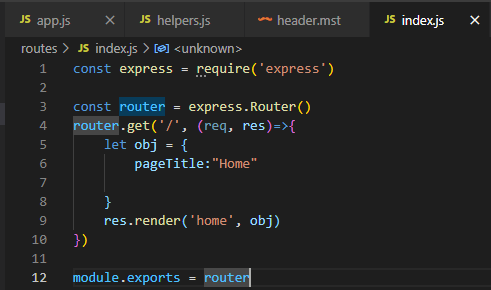
app.js:



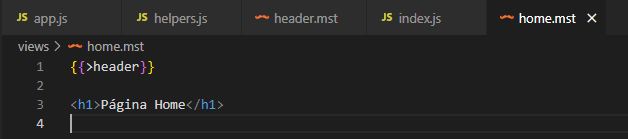
header.mst:



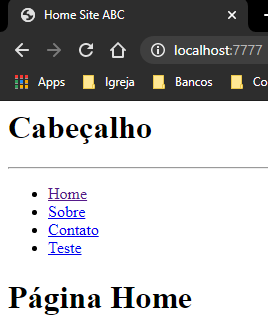
index.js:



home.mst:



Navegador:



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# Aula 09 – Estrutura MVC no Node

Padrão de arquitetura MVC:

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

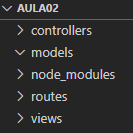
**V → View** → Responsável pela parte Visual (mustache)

**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.

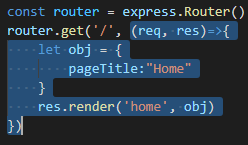
A pasta views, nós já temos, portanto, criaremos agora as pastas models e controllers, na raiz do nosso projeto:



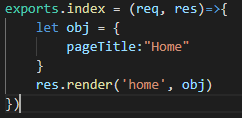
**controllers** → É criado um arquivo controller para cada parte específica do nosso sistema. Iniciaremos criando o arquivo **homeController.js** dentro da pasta controllers.

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:



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:



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:

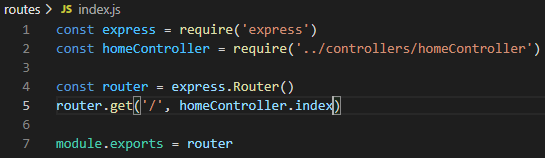
**const homeController = require('../controllers/homeController')**

e o parâmetro:

**homeController.index**

no **router**

index.js fica assim:



\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

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:

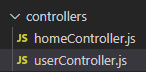
Para isto, criamos a variável userController:

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

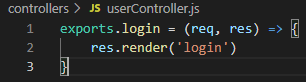
E criamos também a sua rota:

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:



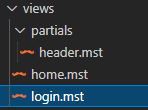
E adicionamos o exports no userController:

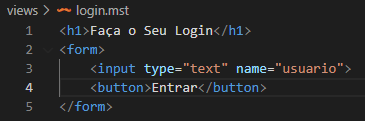


Agora, para que a página login, possa ser efetivamente renderizada, precisamos criá-la.

Dentro da pasta views, criamos o arquivo login.mst.

Criamos um arquivo simples, apenas para teste:



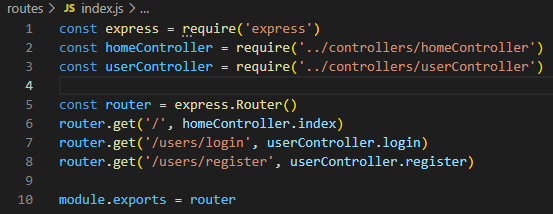


HTML:

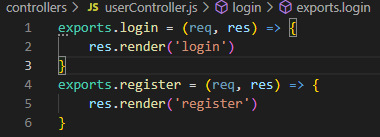


No final da aula, criamos uma rota para registro de usuários, apenas para visualizarmos a facilidade que este sistema de MVC trás para o desenvolvimento do sistema.

Index.js:



userController.js:



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# 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.

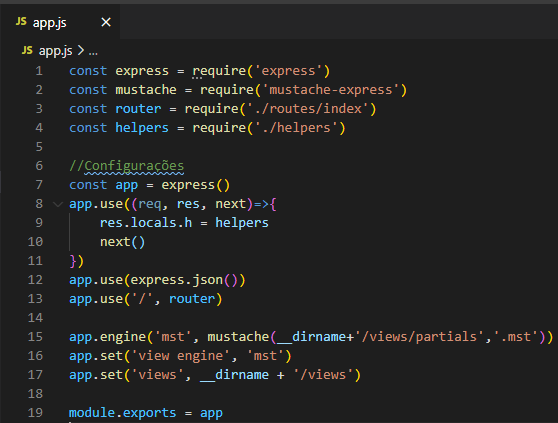
Existem 2 tipos de middleware no Node:

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

- Local → Específico em uma aplicação.

Primeira interferência no código:

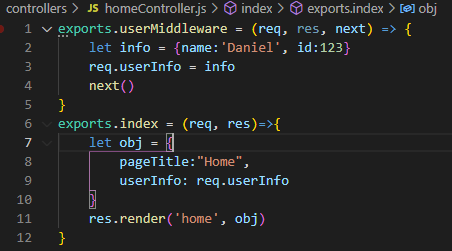
No app.js, trocamos o **app.use(express.json())** de lugar com o **app.use(‘/’, router)** estes são middleware globais que já estávamos utilizando e de acordo com a orientação do Bonieky, o ideal é que o **router** fique por último entre os middleware. Assim:



Vamos agora criar um middleware local:

Este middleware irá adicionar informações em uma requisição de login.

Vamos criar no homeController.js uma nova função:



**exports.userMiddleware = (req, res, next) => { →** userMiddleware → Nome sugestivo

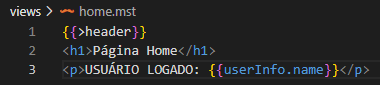
**let info = {name:'Daniel', id:123} →** Estamos utilizando esta linha para simular a coleta destas informações como se fosse de um banco de dados.

**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)

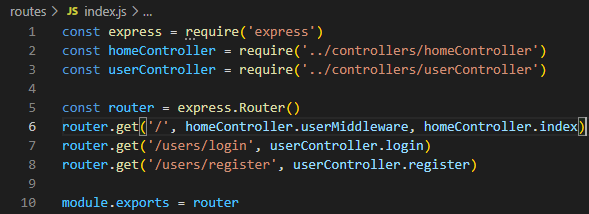
**home.mst**:



<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**:



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