

Boas-vindas!

Esteja confortável, pegue uma água e se acomode em um local tranquilo que já começamos.

Como você **chega?**

1



2



3



Esta aula será

- gravada

Resumo

da última aula:

- ✓ Revisamos o que é o Node.js e seu uso no back-end
- ✓ Entendemos a diferença entre um módulo nativo e um módulo de terceiros
- ✓ Conhecemos a função do NPM e o processo de instalação da dependência
- ✓ Conhecemos o processo de atualização de dependências.

Perguntas?

Aula 06. BACKEND

Servidores Web

Objetivos da aula

- Entenda o protocolo HTTP
- Aprenda a criar um servidor usando o módulo http nativo
- Crie um servidor express
- Saiba mais sobre as solicitações GET de um servidor express.

Vamos revisar os **principais**
pontos da aula anterior?

Glossário – Aula 5

Nodejs: Ambiente de execução Javascript. É o coração dos nossos programas.

Módulo: Conjunto de funções e objetos que permitem resolver um problema específico.

Módulo nativo: Módulos que são inclusos na instalação do nodejs, não requerem instalação, basta importar.

Dependências: Todos os módulos externos ao nosso projeto que precisamos instalar para executar o projeto.

Dependência Local: Dependências instaladas apenas dentro do nosso projeto.

Dependência global: Dependências que são instaladas no nível do computador e podem ser usadas em qualquer projeto.

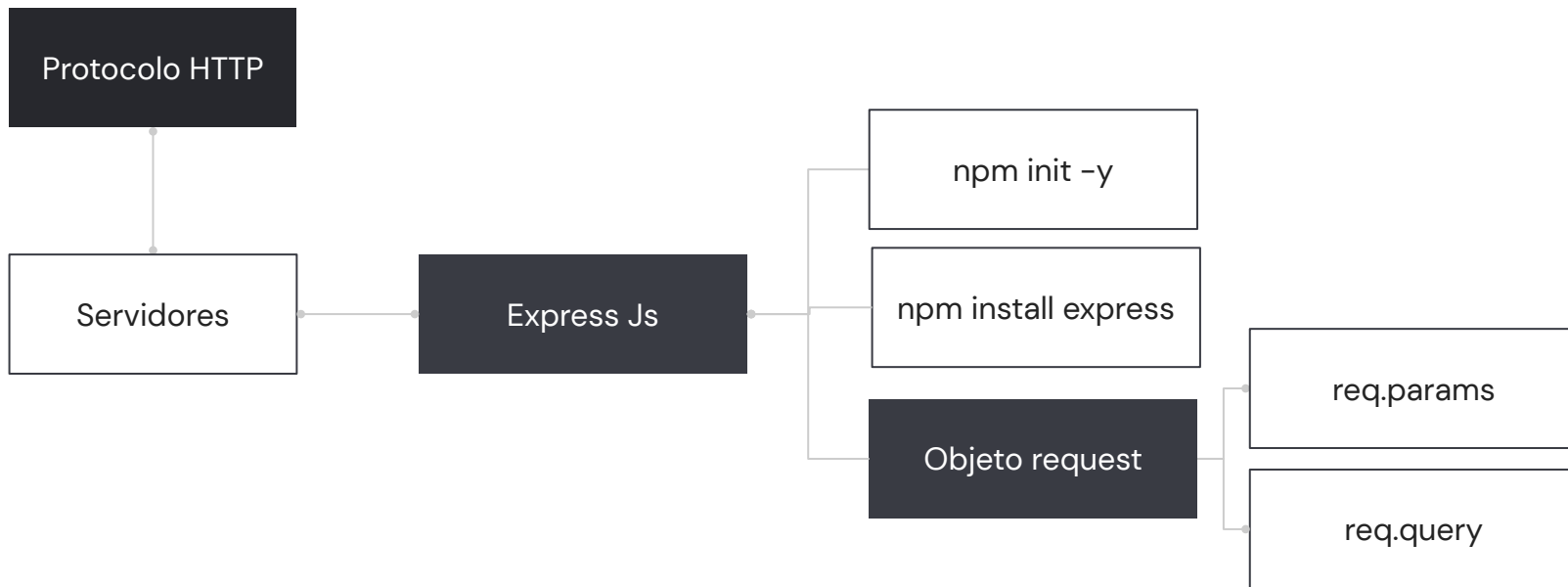
Glossário – Aula 5

package.json: Arquivo de especificação do projeto, utilizado para conter as dependências a serem utilizadas no projeto.

Versão: Cada uma das etapas e mudanças que nosso projeto representa, pode ser maior, menor ou patch.

Entendendo o protocolo HTTP

MAPA DE CONCEITOS



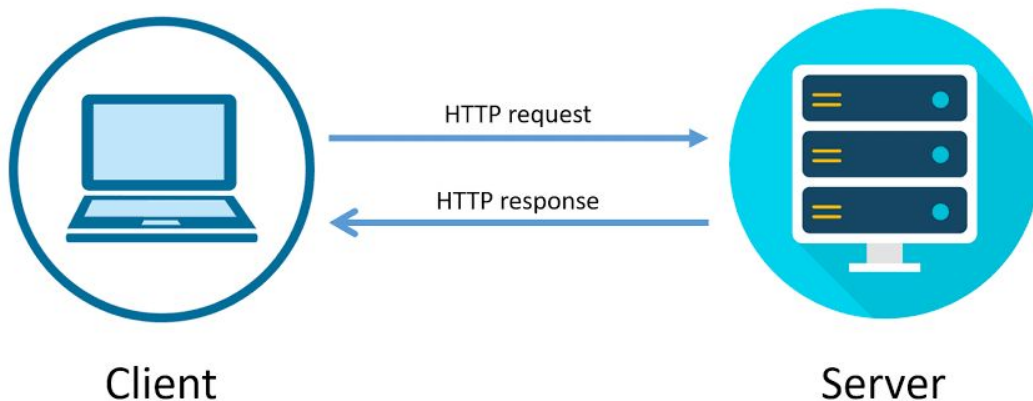
HTTP

HTTP (**Hyper Text Transfer Protocol**) refere-se a um protocolo que é um conjunto de regras que permite a comunicação entre dois ou mais sistemas. Graças a este protocolo, os computadores sabem como se comunicar entre si e permitem que eles se comuniquem com os servidores para obter dados. (podemos vê-lo em todas as páginas que visitamos).

[Leitura complementar sobre o protocolo HTTP](#)

HTTP

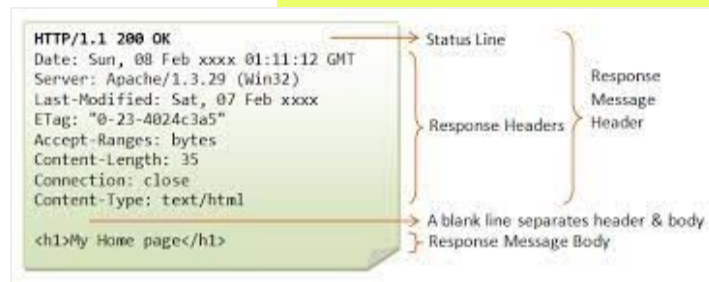
Como funciona? Baseia-se em um modelo de solicitação - resposta .
Então o cliente tem que fazer um pedido de informação, então o servidor se encarrega de responder com essa informação.



Que tipo de informação solicitamos?

Ao **fazer uma solicitação**, estamos solicitando determinados recursos do servidor, que podem ser:

- ✓ Um pedaço de informação como um nome, uma data, uma idade...
- ✓ Informações mais complexas como uma imagem, um vídeo.
- ✓ Um arquivo para baixar
- ✓ Mesmo uma página web inteira!



Exemplo de resposta do servidor, retornando uma tag h1

Lida com várias solicitações

Quando programamos nosso servidor, fazemos isso para ouvir solicitações, a pergunta é: **solicitações de quem ouvir?**

Sob sua configuração padrão, **um servidor pode ouvir várias solicitações de vários clientes ao mesmo tempo!**

E se o servidor cair?

Sabemos que um programa começa, faz suas operações e depois termina.

O que há de diferente no servidor?

Como você pode receber solicitações em diferentes períodos de tempo sem finalizar? 🤔

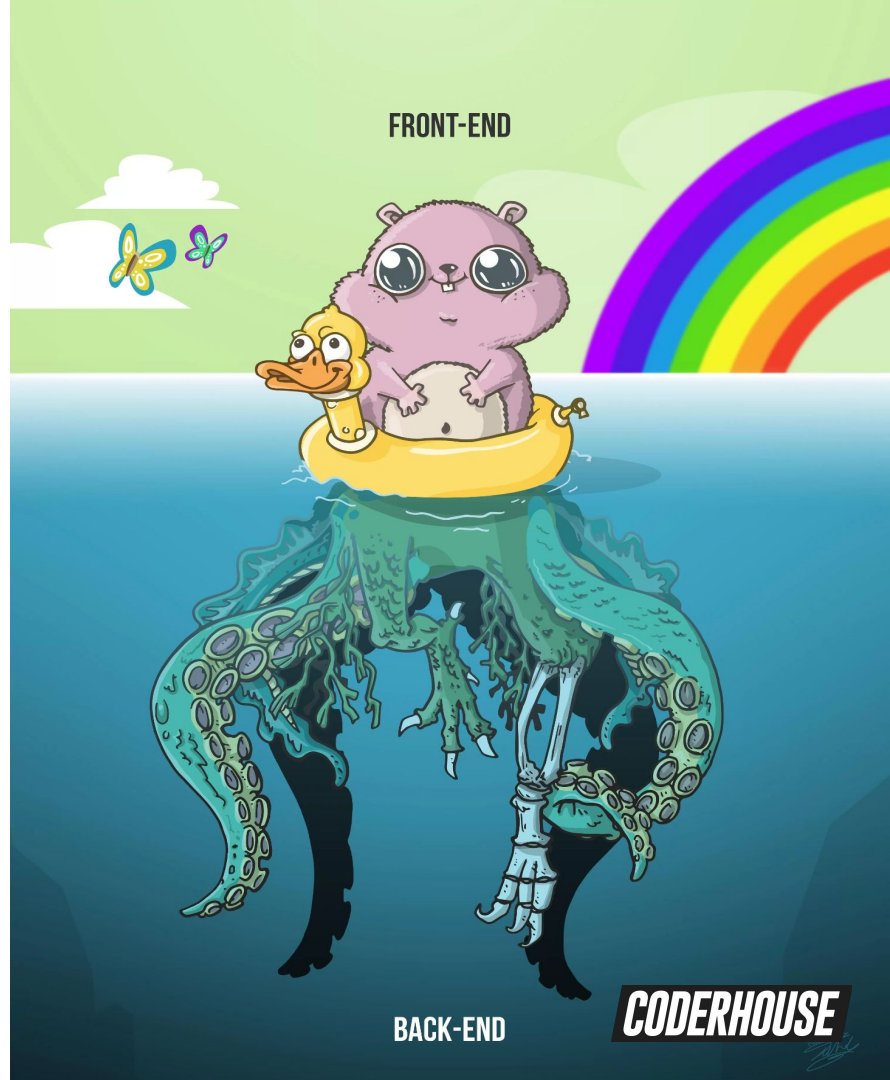
Importante!

O cliente é sempre quem faz as requisições (requests) e o servidor sempre será quem faz as respostas (responses). **Quando fazemos frontend, somos clientes.** Neste curso você terá que ser o servidor e devolver as respostas!

Para lembrar

Enquanto o **FrontEnd** é focado na experiência do usuário, que seja o mais apresentável e amigável possível!

...o **BackEnd** tem uma estrutura interna para que tudo funcione corretamente. O usuário não precisa estar ciente.



O que precisamos antes
de começar?

Instalar nodemon globalmente

Como nosso servidor fica ouvindo o tempo todo, **as alterações que fazemos no código não são refletidas automaticamente**. Portanto, temos que desligar o servidor e ligá-lo novamente .

Só assim podemos visualizar as alterações que fazemos no código (mesmo que seja um simples `console.log()`)

O **Nodemon** nos **permitirá reiniciar automaticamente o servidor assim que detectar que há alterações no código**. Dessa forma, podemos nos concentrar no código, sem precisar reiniciar manualmente toda vez que quisermos ver algo.

Para instalá-lo, basta executar o comando:

```
npm install -g nodemon
```

[Documentação Nodemon](#)



Exemplo ao vivo

- ✓ Crie um servidor com o módulo nativo Nodejs "http". Configure uma resposta que contenha a mensagem "My first hello world from the backend!"
- ✓ O servidor deve escutar na porta 8080. (Execute com nodemon)
- ✓ Teste o servidor a partir do navegador.
- ✓ Faça algumas alterações no código e verifique se ele reinicia automaticamente.

1

Implementando o servidor

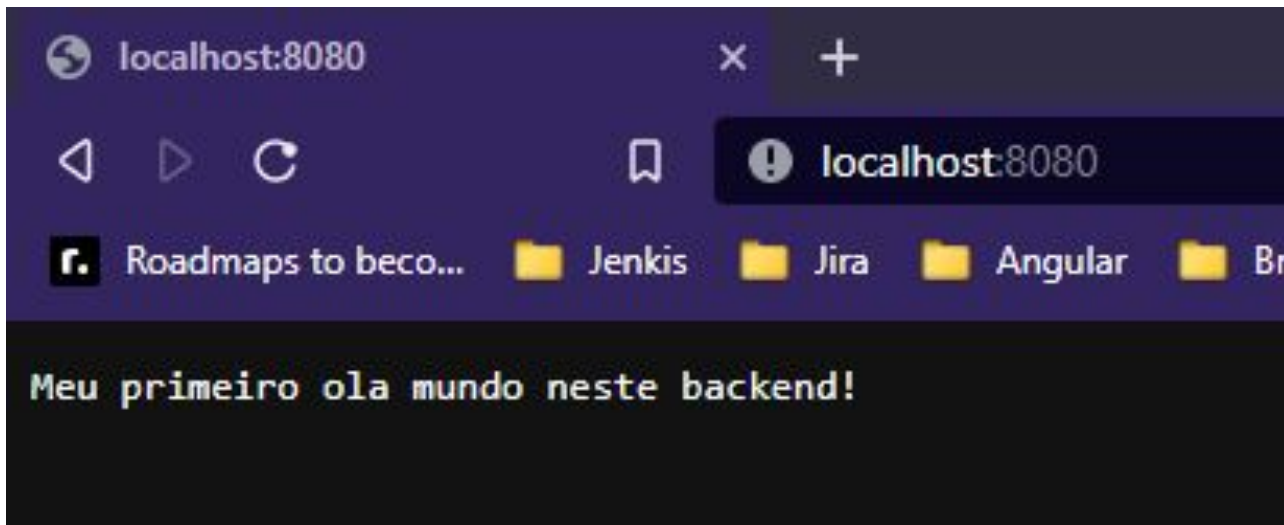
```
const http = require('http');

const server = http.createServer((request, response) => {
  response.end("Meu primeiro olá mundo neste backend!");
});

server.listen(8080, () => {
  console.log("Listening on port 8080");
});
```

2

Testando a partir do navegador



Servidor com express js

O que é Express js?

Express js é um **framework minimalista** que **permite desenvolver servidores mais complexos**.

Isso nos fornecerá:

- ✓ Use rotas diferentes para solicitações.
- ✓ Melhorar a estrutura do nosso projeto.
- ✓ Lidar com funcionalidades mais complexas e uso de middlewares.



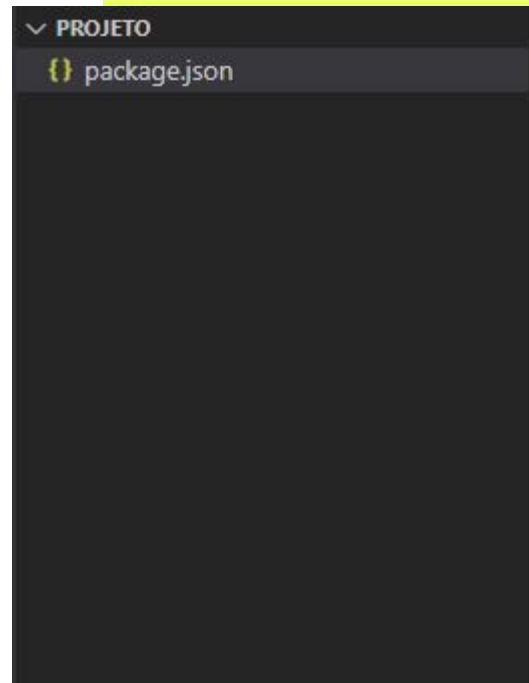
[Documentação Express js](#)

Vamos preparar um projeto Express js

1 npm init -y

O **Express** não é nativo do **nodejs**, portanto, primeiro precisamos ter um `package.json` para gerenciar as dependências a serem instaladas.

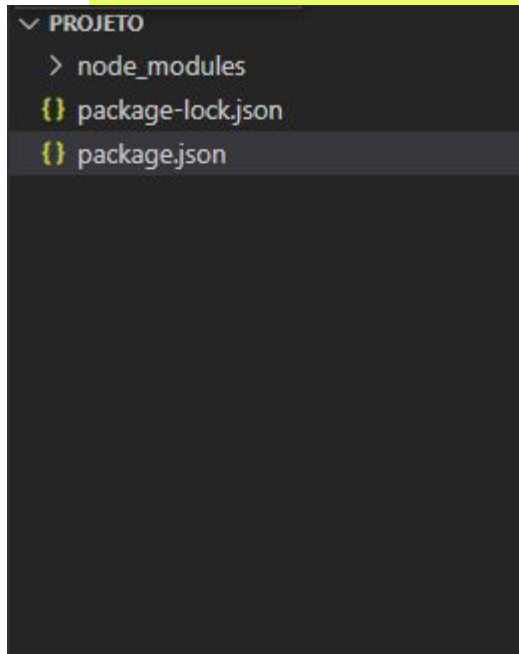
Assim que tivermos o `package.json` em nossa pasta, podemos continuar instalando as dependências.



2 npm install express

Prosseguimos para **instalar o express js localmente**. Ao executar este comando, notamos como é gerada uma pasta `node_modules`, que é onde o `express` é armazenado.

A partir deste ponto já temos a estrutura elementar instalada, o resto é mais "flexível".



3

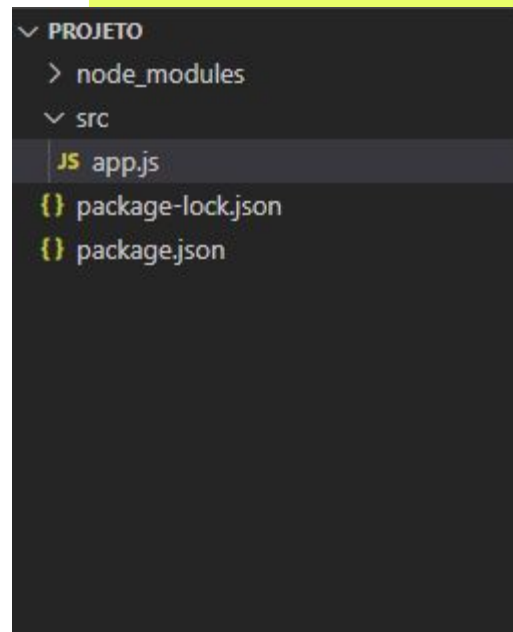
Estruturando o projeto

É recomendável ter uma pasta **src**, onde ficará todo o nosso código, dentro da qual criaremos um arquivo com o nome "app.js"

Por fim, o arquivo app.js agora pode importar a dependência do express js instalada , seja por commonjs: `const express = require('express');`

ou por módulo (lembre-se de colocar o tipo:"module" em package.json):

```
import express from 'express';
```



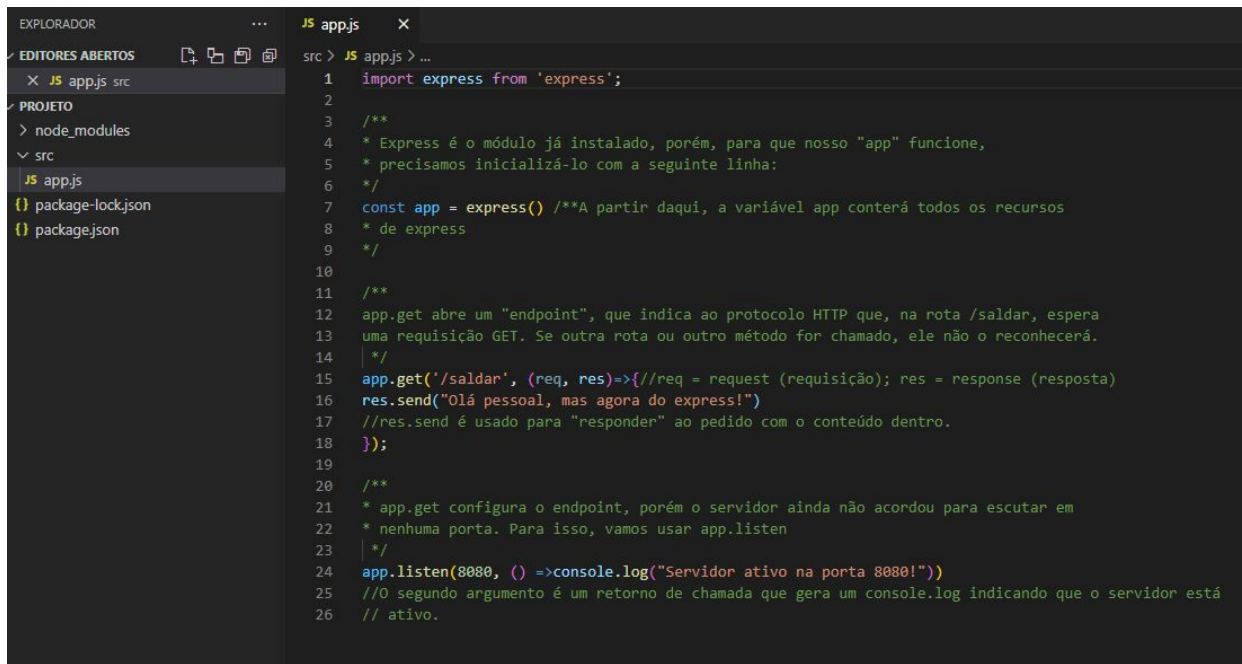


Exemplo ao vivo: consulta no Express

- ✓ Estruture um servidor baseado em express, que escuta solicitações na porta 8080
- ✓ Crie uma função para o método GET na rota '/saldar', que responderá com "Olá pessoal, mas agora do express!"
- ✓ Execute com nodemon e teste o endpoint gerado no navegador

1

Implementando o projeto

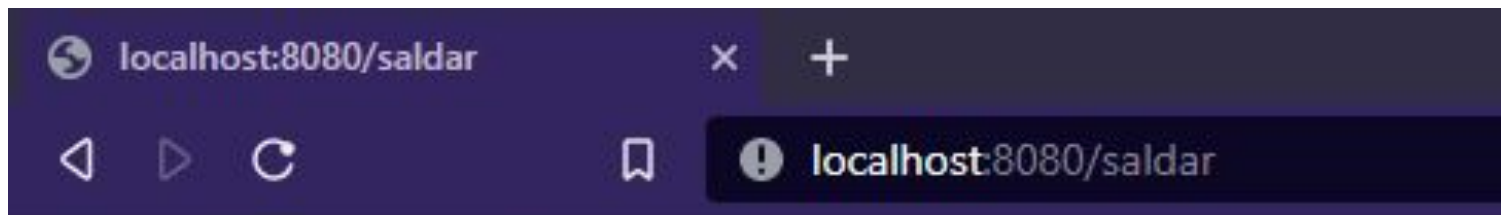


The screenshot shows a code editor with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with a 'src' directory containing 'app.js'. The code editor shows the content of 'app.js', which is a JavaScript file using Express.js to create a simple web server. The code includes comments in Portuguese explaining the steps: importing Express, initializing the app, defining a GET endpoint for '/saldar', and listening on port 8080.

```
1 import express from 'express';
2
3 /**
4  * Express é o módulo já instalado, porém, para que nosso "app" funcione,
5  * precisamos inicializá-lo com a seguinte linha:
6  */
7 const app = express() /**A partir daqui, a variável app conterá todos os recursos
8  * de express
9  */
10
11 /**
12  * app.get abre um "endpoint", que indica ao protocolo HTTP que, na rota /saldar, espera
13  * uma requisição GET. Se outra rota ou outro método for chamado, ele não o reconhecerá.
14  */
15 app.get('/saldar', (req, res)=>{req = request (requisição); res = response (resposta)
16 res.send("Olá pessoal, mas agora do express!")
17 //res.send é usado para "responder" ao pedido com o conteúdo dentro.
18 });
19
20 /**
21  * app.get configura o endpoint, porém o servidor ainda não acordou para escutar em
22  * nenhuma porta. Para isso, vamos usar app.listen
23  */
24 app.listen(8080, () =>console.log("Servidor ativo na porta 8080!"))
25 //O segundo argumento é um retorno de chamada que gera um console.log indicando que o servidor está
26 // ativo.
```

2

Testando no navegador no endpoint especificado



Olá pessoal, mas agora do express!

Outras respostas com express

Conhecendo outros tipos de elementos com os quais podemos responder com express

- ✓ Crie um projeto baseado em express js, que possui um servidor que escuta na porta 8080. Além de configurar os seguintes endpoints:
- ✓ O endpoint do método GET para a rota `/bemvindo` deve retornar um html com letras na cor azul, em uma string, welcome.
- ✓ O endpoint do método GET para a rota `/usuario` deve retornar um objeto com os dados de um usuário falso: `{nome, sobrenome, idade, email}`



Break

5 minutos e voltamos!





Break

10 minutos e voltamos!



Objeto request

Objeto request

Ao fazer os **endpoints**, até agora **usamos apenas o elemento "res"**, que usamos para **responder a uma solicitação**.

No entanto, nestes últimos exercícios deixamos de fora o **"req"** que vem à esquerda, agora vamos abordar para que serve e como podemos utilizá-lo.

[Documentação de express js request](#)

O **objeto req** tem três propriedades principais: **req.query**, **req.params**, **req.body**.

Hoje vamos cobrir **req.query** e **req.params**, na próxima aula vamos nos aprofundar em **req.body**.

Instalar nodemon globalmente

Como nosso servidor fica ouvindo o tempo todo, **as alterações que fazemos no código não são refletidas automaticamente**. Portanto, temos que desligar o servidor e ligá-lo novamente.

Só assim podemos visualizar as alterações que fazemos no código (mesmo que seja um simples `console.log()`)

O **Nodemon** nos **permitirá reiniciar automaticamente o servidor assim que detectar que há alterações no código**. Dessa forma, podemos nos concentrar no código, sem precisar reiniciar manualmente toda vez que quisermos ver algo.

Para instalá-lo, basta executar o comando: `npm install -g nodemon`

[Documentação Nodemon](#)

req.params

req.params

É utilizado quando precisamos obter elementos dinâmicos da rota que o cliente está chamando.

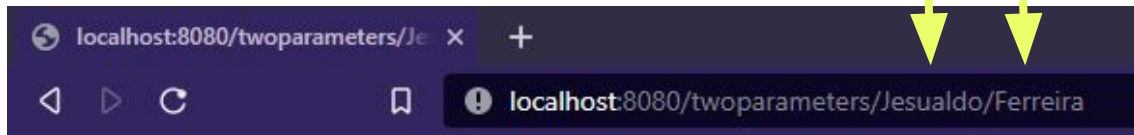
Para definir um "parâmetro" dentro da rota a trabalhar, basta colocar o símbolo de dois pontos : antes do parâmetro, desta forma, express reconhece que queremos que aquele elemento seja dinâmico.

[Documentação Express js req.param](#)

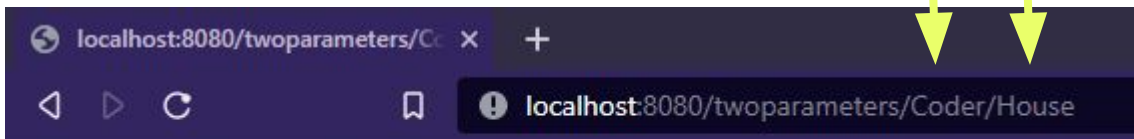
Definindo endpoints com parâmetros

```
JS endpointParametro.js X
src > JS endpointParametro.js > ...
1  import express from 'express';
2
3  const app = express();
4  /**
5   * Usamos os dois pontos ( : ) para indicar que queremos que esse seja o parâmetro.
6   * Isso nos permite criar uma rota dinâmica que pode receber qualquer parâmetro, em vez
7   * de tentar adivinhar. Agora podemos inserir qualquer nome do url, em vez
8   * de ter que cadastrar 10000000000 rotas com 10000000000 nomes diferentes
9   */
10 app.get('/oneparameter/:name', (req, res)=>{
11   //:o parâmetro agora estará dentro do objeto req.params
12   console.log(req.params.name)
13   res.send(`Bem-vindo, ${req.params.name}`)
14 });
15
16 app.get('/twoparameters/:name/:lastname', (req, res)=>{
17   //:name e :lastname agora estarão dentro do objeto req.params
18   // Podemos definir em nosso endpoint quantos parâmetros precisarmos!
19   res.send(`O nome completo é: ${req.params.firstname} ${req.params.lastname}`)
20 });
21
22 app.listen(8080, () => console.log("Pronto para receber solicitações!"));
```

Testando endpoint de dois parâmetros, passando nomes dinamicamente



O nome completo é: Jesualdo Ferreira



O nome completo é: Coder House



Exemplo ao vivo: Caso de uso com Parâmetros

- ✓ Dado um array de objetos do tipo usuário, crie um servidor em express que permita obter os referidos usuários.
- ✓ O caminho raiz '/' deve retornar todos os usuários
- ✓ a rota `/:userId` deve retornar apenas o usuário com esse id.

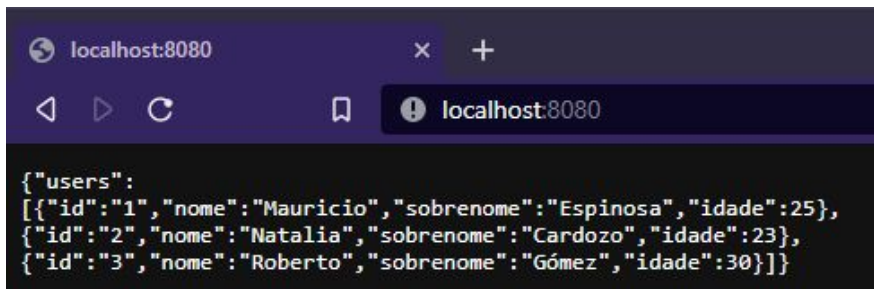
1

Exemplo de implementação

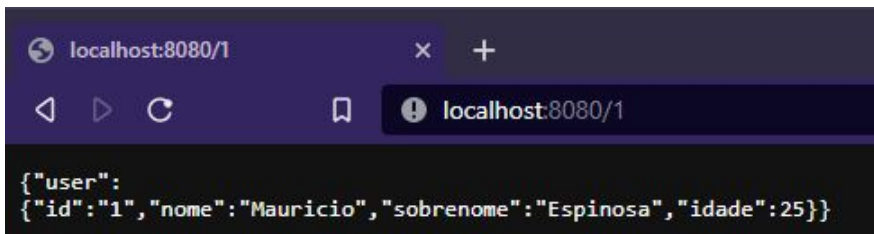
```
JS casoPraticoParametros.js X
src > JS casoPraticoParametros.js > ...
1 import express from 'express';
2 const app = express();
3 const users = [
4   {id:"1",nome: "Mauricio", sobrenome: "Espinosa", idade:25},
5   {id:"2",nome: "Natalia", sobrenome: "Cardozo", idade: 23},
6   {id:"3",nome: "Roberto", sobrenome: "Gómez", idade: 30}
7 ];
8
9 app.get('/', (req, res)=>{
10   res.send({users}) //É altamente recomendável enviar os dados em formato de objeto em vez de
11   //enviá-los como um único array, isso permite que, se formos inserir mais informações no futuro
12   //nós não precisaremos alterar o tipo de resposta do lado do cliente.
13 });
14
15 app.get('/:idUser', (req, res)=>{
16   let userid = req.params.userid; // Obtém o id do usuário para funcionar
17   /**
18    * Prosseguimos em busca do usuário que tem o id passado por params.
19    */
20   let user = users.find(u=>u.id==idUser);
21   //Se não encontrar o usuário, deve encerrar a função retornando um erro
22   if(!user) return res.send({error: "Usuário não encontrado"});
23   //Caso a função não tenha sido finalizada, significa que o usuário foi encontrado.
24   res.send({usuario})
25 });
26
27 app.listen(8080, () =>console.log("Pronto para testar caso de uso"))
```

2 Testes

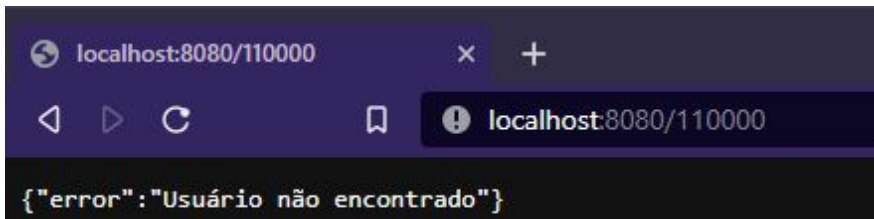
- ✓ **Teste de rota base:** deve retornar todos os usuários.
- ✓ **Teste com parâmetro válido:** Deve encontrar o usuário e retornar apenas esse usuário.
- ✓ **Teste com parâmetro inválido:** Deve gerar um erro ao não encontrar um usuário com esse id.



```
{
  "users": [
    {
      "id": "1",
      "nome": "Mauricio",
      "sobrenome": "Espinosa",
      "idade": 25
    },
    {
      "id": "2",
      "nome": "Natalia",
      "sobrenome": "Cardozo",
      "idade": 23
    },
    {
      "id": "3",
      "nome": "Roberto",
      "sobrenome": "Gómez",
      "idade": 30
    }
  ]
}
```



```
{
  "user": {
    "id": "1",
    "nome": "Mauricio",
    "sobrenome": "Espinosa",
    "idade": 25
  }
}
```



```
{
  "error": "Usuário não encontrado"
}
```

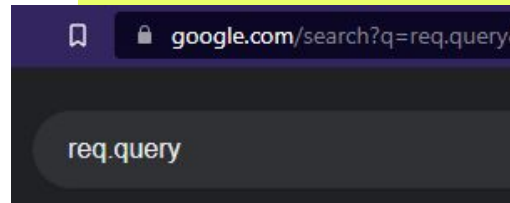
req.query

req.query

Como o próprio nome indica, **query** refere-se às **múltiplas consultas que podem ser feitas a um determinado endpoint**, basta que na url coloquemos o símbolo **?**, o express reconhecerá que as informações precisam ser passadas para o objeto `req.query` para usá-las no endpoint.

Quando pesquisamos algo em nosso navegador, chamamos um endpoint fazendo uma determinada consulta.

[Documentação express js req.query](#)





Importante!

Conforme o dinamismo das urls aumenta, é **importante configurar o servidor para receber dados complexos da url**, então você deve usar a linha:

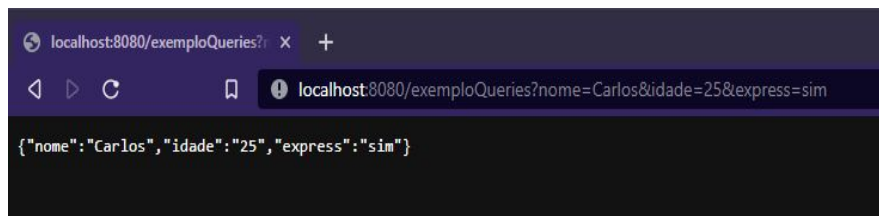
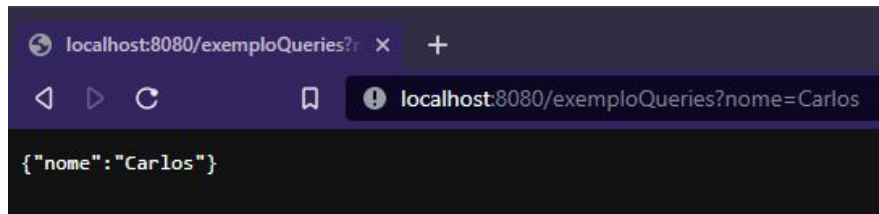
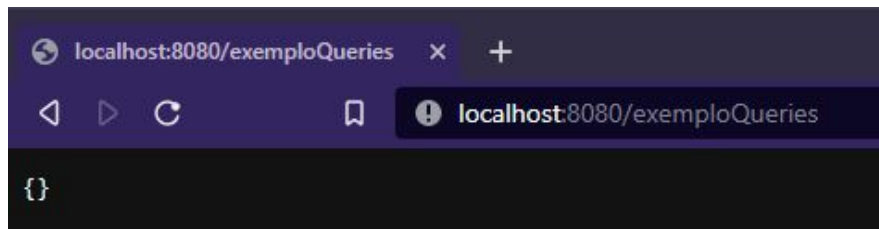
```
app.use(express.urlencoded({extended:true}))
```

A linha acima **permitirá** que o servidor interprete melhor os dados complexos que trafegam da url e os mapeie corretamente no req.query

Definição da estrutura de um endpoint com req.query

```
JS exemploQuery.js X
src > JS exemploQuery.js > ...
1  import express from 'express';
2
3  const app = express();
4
5  app.use(express.urlencoded({extended:true}));
6
7  app.get('/exemploQueries', (req, res)=>{
8    /**
9     * Observe algo interessante: Aqui não é necessário antecipar o parâmetro que o cliente deve
10     * inserir. Simplesmente chamando o endpoint, o cliente pode
11     * inserir as consultas que você precisa no url com o símbolo ?
12     */
13     let consultas = req.query;
14     /**
15     * Ao contrário de req.params, aqui não contemplei que tipo de coisas podem ser solicitadas a mim,
16     * embora possamos delimitá-lo fazendo uma desestruturação
17     */
18     let {nome, sobrenome, idade} = req.query;
19     /**
20     * Então, não importa o que volte da consulta, só vou extrair o nome, sobrenome e idade
21     * Isso aumenta a segurança do servidor porque evitamos receber elementos estranhos de
22     * a url
23     */
24     res.send(consultas)
25   });
26
27  app.listen(8080, ()=>console.log("Pronto para testar consultas"))
```

- ✓ Se chamarmos o endpoint sem query, notamos que o objeto req.query está vazio
- ✓ Se chamarmos o mesmo endpoint, agora com o símbolo ? para definir um query param, observamos como ele agora aparece no objeto
- ✓ Por fim, podemos colocar quantas queries quisermos usando o símbolo & e notamos que o objeto req.query possui mais de um parâmetro de consulta para poder ser utilizado.





Exemplo ao vivo: Caso de Uso Req.query

- ✓ Dado um array de objetos do tipo user, vamos filtrar por gênero
- ✓ O caminho raiz '/' deve retornar todos os usuários, mas agora vamos colocar um parâmetro de query com ?, indicando que queremos um gênero específico. no caso de envio sem query, deverá retornar todos os usuários.

Exemplo: implementação endpoint que usa req.query

```
JS casoPraticoQuery.js X
src > JS casoPraticoQuery.js > ...
1 import express from 'express';
2
3 const app = express();
4
5 app.use(express.urlencoded({extended: true}));
6
7 //Filtraremos por gênero masculino (M) e feminino (F)
8 const users = [
9   {id:"1",nome: "Dalia", sobrenome: "Gómez", genero: "F"},
10  {id:"2",nome: "Myrna", sobrenome: "Flores", genero: "F"},
11  {id:"3",nome: "Armando", sobrenome: "Mendoza", genero: "M"},
12  {id:"4",nome: "Dalia", sobrenome: "Gómez", genero: "F"},
13  {id:"5",nome: "Herminio", sobrenome: "Fuentes", genero: "M"},
14  {id:"6",nome: "Juan", sobrenome: "Zepeda", genero: "M"},
15 ];
16
17 app.get('/', (req, res)=>{
18   let genero = req.query.genero;
19   //Se o gênero não foi inserido, ou o gênero não é nem M nem F, o filtro é inválido.
20   if(!genero || (genero!="M"&&genero!="F")) return res.send({users})
21   // Caso contrário, continuamos com o processo de filtragem.
22   let filterUsers = users.filter(user=> user.genero==genero);
23   res.send({users:filterUsers});
24 });
25
26 app.listen(8080, () =>console.log("Pronto para fazer filtros"));
```

1. Chamar o endpoint sem query param permite que todos os usuários retornem sem filtrar
2. Enviando a query com gênero = F, notamos como ela retorna apenas usuários com gênero F.
3. O mesmo caso com M.

 Acabamos de criar um filtro de usuário perfeitamente funcional!

1

```
localhost:8080
{"users":
  [{"id": "1", "nome": "Dalia", "sobrenome": "Gómez", "genero": "F"},
  {"id": "2", "nome": "Myrna", "sobrenome": "Flores", "genero": "F"},
  {"id": "3", "nome": "Armando", "sobrenome": "Mendoza", "genero": "M"},
  {"id": "4", "nome": "Dalia", "sobrenome": "Gómez", "genero": "F"},
  {"id": "5", "nome": "Herminio", "sobrenome": "Fuentes", "genero": "M"},
  {"id": "6", "nome": "Juan", "sobrenome": "Zepeda", "genero": "M"}]}
```

2

```
localhost:8080/?genero=F
{"users":
  [{"id": "1", "nome": "Dalia", "sobrenome": "Gómez", "genero": "F"},
  {"id": "2", "nome": "Myrna", "sobrenome": "Flores", "genero": "F"},
  {"id": "4", "nome": "Dalia", "sobrenome": "Gómez", "genero": "F"}]}
```

3

```
localhost:8080/?genero=M
{"users":
  [{"id": "3", "nome": "Armando", "sobrenome": "Mendoza", "genero": "M"},
  {"id": "5", "nome": "Herminio", "sobrenome": "Fuentes", "genero": "M"},
  {"id": "6", "nome": "Juan", "sobrenome": "Zepeda", "genero": "M"}]}
```

Qual é a diferença com params?

A principal diferença entre **req.params** e **req.query** é que em **req.query** eu posso inserir o número de consultas que eu quiser, já que as consultas não estão incluídas na rota, mas são um elemento separado.

Então, se eu não souber a quantidade de coisas que vão ser consultadas na minha rota, a melhor opção é usar queries , enquanto **se eu precisar apenas de um número específico e reduzido de parâmetros, teria que optar por params.**

Afinal, não existe um melhor que o outro, eles atendem a casos diferentes e podemos até usar os dois na mesma consulta.

Servidor com express

Vamos relembrar

Aula 4: Introdução ao Nodejs

Pegamos a classe ProductManager que tinha persistência na memória

Desenvolvemos um ProductManager baseado em arquivo com métodos assíncronos

Conseguimos mudar a persistência de memória para arquivos

Perguntas?

Como foi a aula?

1

Que bom

O que foi super legal na aula e podemos sempre trazer para as próximas?

2

Que pena

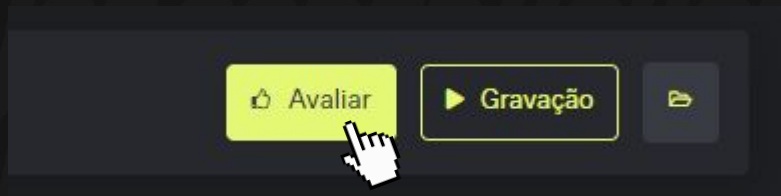
O que você acha que não funcionou bem e precisamos melhorar?

3

Que tal

Qual sugestão deveríamos tentar em próximas aulas?

O que você achou da aula?



Seu feedback vale pontos para o Top 10!! 🕶️



Deixe sua opinião!

1. Acesse a plataforma
2. Vá na aula do dia
3. Clique em **Avaliar**



Servidor com express

DESAFIO OBRIGATÓRIO



DESAFIO OBRIGATÓRIO

Servidor com express

Objetivo:

Desenvolva um servidor baseado em express onde possamos consultar nosso arquivo de produtos.

Aspectos a incluir:

- ✓ Você precisará usar a classe ProductManager que usamos atualmente com persistência de arquivo.
- ✓ Desenvolva um servidor expresso que, em seu arquivo app.js, importe o arquivo ProductManager que temos atualmente.

O servidor deve ter os seguintes endpoints:

- ✓ Caminho '/products', que deve ler o arquivo products e retorná-los dentro de um objeto. Adicione suporte para receber por parâmetro de query o valor ?limit= que receberá um limite de resultados.
 - Se nenhuma consulta de limite for recebida, todos os produtos serão devolvidos
 - Se for recebido um limite, devolva apenas o número de produtos solicitados



DESAFIO OBRIGATÓRIO

Servidor com express

Aspectos a incluir:

- ✓ rota '/products/:pid', que deve receber o pid (ID do produto) por req.params, e retornar apenas o produto solicitado, ao invés de todos os produtos.

Sugestões:

- ✓ Sua classe lê arquivos com promessas. lembre-se de usar async/await em seus endpoints
- ✓ Use um arquivo que já tenha produtos, pois o desafio é apenas para gets.

Formato de entrega:

- ✓ Link para o repositório Github com o projeto completo, que deve incluir:
 - src com app.js dentro e seu ProductManager dentro.
 - package.json com as informações do projeto.
 - NÃO INCLUA node_modules gerados.

Resumo

da aula de hoje:

- ✓ Entendemos o protocolo HTTP
- ✓ Aprendemos a criar um servidor usando o módulo http nativo
- ✓ Criamos um servidor express
- ✓ Aprendemos mais sobre as solicitações GET de um servidor express.



**Obrigado por estudar
conosco!**