

```

const express = require('express');
const app = express();
app.use(express.urlencoded({extended: true}));
//Express usado para interpretar dados enviados via formulários HTML (método POST).
// Especificamente, ela faz o parsing de dados no formato application/x-www-form-urlencoded,
// que é o formato padrão quando você envia um formulário simples.

//extended: true #Permite enviar dados mais complexos, como objetos aninhados.
//extended: false #Só permite dados simples (chave-valor)

app.get('/', (req, res) => {
  res.send(`<form action="/thanks" method="POST">
    <label for="user-name">Your name</label><input name="user[name]" id="user-name" placeholder="typing your name">
    <label for="user-age">Your age</label><input name="user[age]" id="user" placeholder="typing your age">
    <button>ok</button>
  `);
});

app.post('/thanks', (req, res) => {
  res.send('<h1>query</h1>');
  console.log(req.body);
  //recebendo requisição via post
});

app.get('/teste/:query?', (req, res) =>{
  console.log(req.query)
  //query(ex): teste/?id=12312&name=jean
})

app.get('/teste/:id?', (req, res) => {
  res.send(req.params.id);
  //parametro: teste/qualquercoisa
});

app.use((req, res) =>{
  res.status(404).send('<h1>404 not found</h1>');
  //Quando o client digitar qualquer coisa, o statusCode irá retornar 404 e com isso, iremos tratar
});

app.listen(3001, () =>{
  console.log('Acessar: http://localhost:3001');
  console.log('Servidor executando');
});

```

```

app.get('/teste/:query?', (req, res) =>{
  console.log(req.query)
  //query(ex): teste/?id=12312&name=jean
})

app.get('/teste/:id?', (req, res) => {
  res.send(req.params.id);
  //parametro: teste/qualquercoisa
});

//o dois pontos(:) depois da barra: signfica que pode ser receber parametro ou query
//'param' é como se fosse uma variavel
//o "?" no final quer dizer que o parametro ou query não é obrigatorio

```

Lembrando que o **app** com ele você cria uma nova instância;
Com **express.function** você estará acessando funções do próprio **express**

{extended: true}

```
Acessar: http://localhost:3001
Servidor executando
{ user: { name: 'sadasd', age: 'sadas' } }
█
```

{extended: false}

```
Acessar: http://localhost:3001
Servidor executando
[Object: null prototype] {
  'user[name]': 'asdsad',
  'user[age]': 'asdasd'
}
█
```

Router and controls

```
aula.js x router.js contato.js
aula.js > ...
1  const express = require('express');
2  const router = require('./router.js')
3  const app = express();
4
5  app.use(express.urlencoded({extended: true}));
6
7  app.use(router);
8
9  app.use((req, res) =>{
10 |   res.status(404).send('<h1>404 not found</h1>');
11 | });
12 app.listen(3001, () =>{
13 |   console.log('Acessar: http://localhost:3001');
14 |   console.log('Servidor executando');
15 | });
16
17 //O use() é um método da aplicação Express
18 // que serve para registrar middlewares ou montar sub-aplicações (como roteadores) em caminhos específicos.
19
```

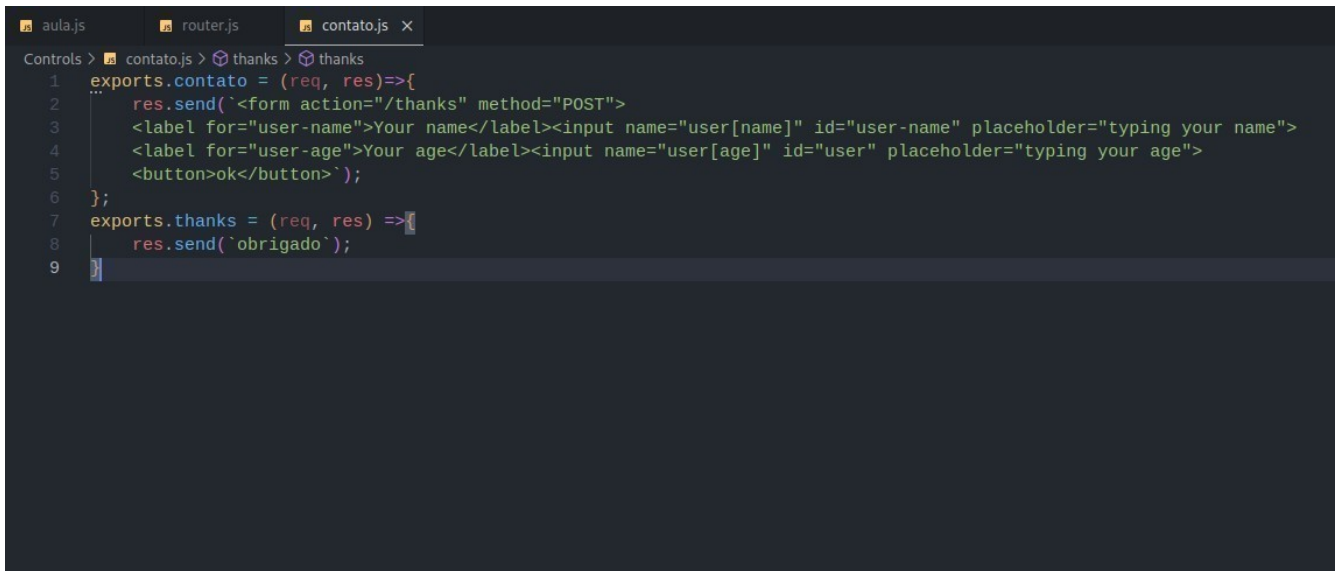
(Arquivo js principal)

Middleware Express são funções que permitem executar ações antes que uma solicitação seja processada no framework web Express.js. Elas são ferramentas que moldam o fluxo de requisição-resposta, melhorando a modularidade do código

```
aula.js router.js x contato.js
router.js > ...
1  const express = require('express');
2  const router = express.Router(); //serve para modular e rotular
3  const form = require('./Controls/contato.js');
4
5  router.get('/form', form.contato);
6  router.post('/thanks', form.thanks);
7
8  module.exports = router;
```

Arquivo js que irá fazer o roteamento de requisições

Imagine que você tem um site com várias seções: /users, /products, /orders. O `express.Router()` permite criar um arquivo separado para cada seção, como se você estivesse "rotulando" cada grupo de rotas. Depois, você monta esses "rótulos" no arquivo controls da aplicação.



```
1 exports.contato = (req, res) => {
2   res.send(`<form action="/thanks" method="POST">
3     <label for="user-name">Your name</label><input name="user[name]" id="user-name" placeholder="typing your name">
4     <label for="user-age">Your age</label><input name="user[age]" id="user" placeholder="typing your age">
5     <button>ok</button>`);
6 };
7 exports.thanks = (req, res) => {
8   res.send(`obrigado`);
9 }
```

Arquivo js que irá tratar das requisições que vai mandar para o cliente

1. Arquivo principal irá chamar o arquivo de router usando o .use()
2. O arquivo de router irá usar uma função de router para rotolar e moldular Usando as funções exportadas de controlls (contato.js) para montar a pagina
3. Arquivo de controlls que irá ser o send e mandar a pagina

Deixando o código mais limpo e organizado

OBS: Dá para você não usar o .router() Porém...

you are creating a new instance of Express (`express()`) and using it as if it were a router. Como o Express é super flexível, isso funciona, mas:

Diferença entre <code>express()</code> e <code>express.Router()</code>	
<code>express()</code> (Instância Completa)	<code>express.Router()</code> (Mini Roteador)
Cria uma nova aplicação Express completa.	Cria uma mini instância só para organizar rotas.
Vem com todas as funcionalidades (middlewares, configs, etc).	Apenas lida com rotas e middlewares locais.
Pode gerar conflitos ou duplicação se usado várias vezes.	Ideal para modularizar sem criar novas instâncias.
Mais pesado se comparado ao Router, pois é uma app completa.	Leve e feito só para modularização.

Por que isso é importante?

Usar `express()` para rotas:

Funciona, mas você está criando várias instâncias completas do Express sem necessidade. Em projetos maiores, isso pode causar problemas como:

Conflitos de configuração entre instâncias.

Desempenho afetado, pois cada "router" é uma aplicação completa.

Usar `express.Router()` para rotas:

É a forma recomendada, porque o Router é leve e feito para modularizar sem criar novas instâncias do Express. Isso deixa o código mais limpo e eficiente.

View e EJS

Você pode criar paginas .html ou renderizar uma página com **view**

```
app.set('views', path.resolve(__dirname, 'views'));  
app.set('view engine', 'ejs');
```

O **.set()** é um método usado para definir configurações específicas para a aplicação Express. Ele é muito usado para configurar opções como diretórios de arquivos de visualização, mecanismo de templates, e até configurações de segurança.

Em outras palavras, o **.set()** no Express é usado para configurar várias opções de como o servidor Express vai se comportar. Ele permite que você defina configurações específicas para sua aplicação, como onde ficam os arquivos de visualização (views), qual o mecanismo de template (view engine) que você está utilizando, entre outras.

parâmetro de **app.set('views', ...)**, que no caso é **'views'**. Essa chave ('views') é apenas um nome usado pelo Express para identificar onde os arquivos de visualização estão localizados. Pode ser outro nome, mas o express vai ficar 'perdido' e só esperar os outros parâmentros para se achar

view engine: O parâmetro **'view engine'** é usado para especificar qual mecanismo de template você está usando para renderizar as views (as páginas HTML que o Express vai enviar como resposta). O Express suporta vários mecanismos de templates, como EJS, Pug, Handlebars, etc.

ejs: O **ejs** é um mecanismo de template que permite incluir JavaScript e lógica dinâmica dentro de arquivos HTML. Ele é muito usado em aplicações Express para gerar páginas dinâmicas com dados que você passa para as views.

Lá no arquivo **controlls.js**, agora você irá usar o **.render('file_name')** que serve para rederenziar o template

```
exports.home = (req, res) =>{  
  res.render('index');  
}
```

Injetando dados em view

Como sabemos, podemos injetar códigos no **view**. Veja um exemplo:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <link rel="stylesheet" href="/assets/css/style.css">
</head>
<body>
  <h1>sal</h1>
  <p>Seja bem-vindo, <%= nome %></p>
</body>
</html>
```

<p>Seja bem-vindo, <%= nome %></p>

Assim que começamos a injetar dados, mas vamos entender em passos. Primeiro temos que criar um variável global lá no `.render()` inserindo como segundo Argumento, veja:

```
res.render('index', {nome: req.session.usuario.nome});
```

O segundo argumento abrimos como um objeto, mas é um objeto 'diferente', A sua propriedade acaba sendo uma chave direta e disponível para ser acessada Diretamente. **Quem faz tudo isso é o próprio .ejs**

O EJS automaticamente disponibiliza as propriedades do objeto passado no segundo argumento do `res.render()`, tornando-as acessíveis diretamente no template.

Por isso que você acessa só colocando a chave

E lá no `.ejs`, para você acessar essas variáveis globais(ou local) você faz uma abertura de `<% ... %>` só que ele recebe alguns caracteres diferentes que dita como irá se comportar essa renderização de códigos


```

1 <% Controle de fluxo (if, for...) %>
2 <%= Imprime escapando caracteres %>
3 <%- Imprime sem escapar caracteres %>
4 <%# Comentário %>
5 <%- include('CAMINHO/ARQUIVO'); %>

```

```

7 <% if (algumacoisa) { %>
8   <%= exibe alguma coisa %>
9 <% } else { %>
10   <%= exibe outra coisa %>
11 <% } %>

```

Um exemplo de que se você pular uma linha você terá que fazer abertura de <%...%>

Sintaxe	Descrição
<%= valor %>	Escapa HTML (protege contra XSS).
<%- valor %>	Não escapa HTML (renderiza código bruto, útil para includes).
<% código %>	Executa código JavaScript sem exibir nada.

Mas o que seria o **include()** ?

Vamos supor que você cortou uma parte do código HTML e tirou o <header> ou <footer> e salvou em um arquivo .ejs. O include() server para você incluir. Ele usa elemento de <%-... por que queremos que ele renderize os códigos

```

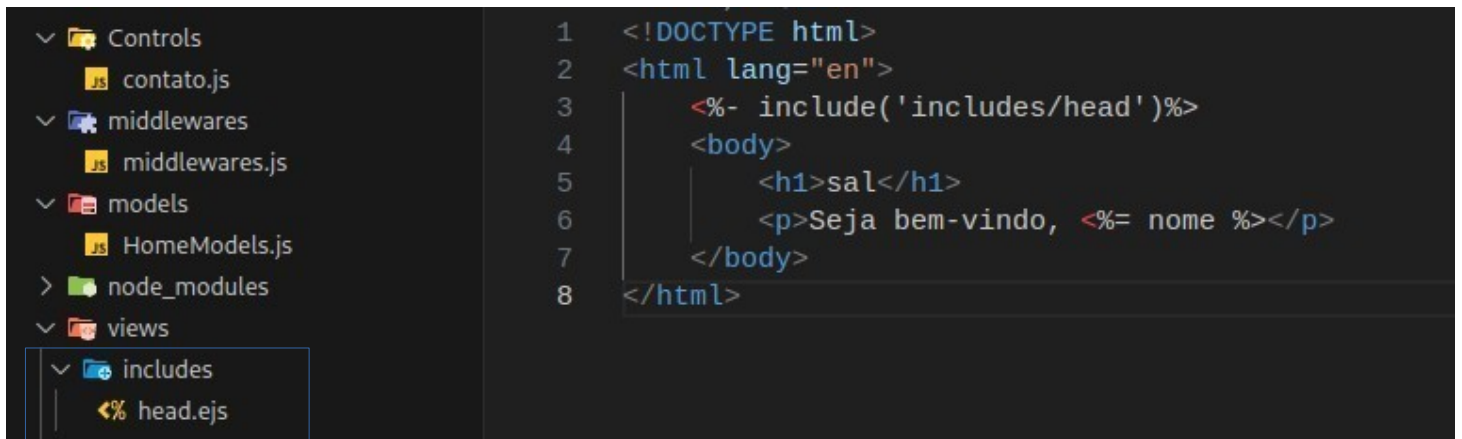
<!DOCTYPE html>
<html lang="pt-br">
<head>
  <title>Meu Site</title>
</head>
<body>
  <%- include('header') %>  <!-- Inclui o cabeçalho -->

  <h1>Bem-vindo ao site!</h1>

  <%- include('footer') %>  <!-- Inclui o rodapé -->
</body>
</html>

```


Você consegue fazer isso até com <head>



The image shows a code editor interface. On the left is a file explorer with a tree view containing the following items:

- Controls
 - contato.js
- middlewares
 - middlewares.js
- models
 - HomeModels.js
- node_modules
- views
 - includes
 - head.ejs

The right pane shows the content of the selected file, `head.ejs`, with the following HTML code:

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <%- include('includes/head')%>
4   <body>
5     <h1>sal</h1>
6     <p>Seja bem-vindo, <%= nome %></p>
7   </body>
8 </html>
```

Arquivos estáticos no Express

O **`app.use(express.static())`** no Express serve para definir uma pasta como pública, permitindo que arquivos estáticos como CSS, imagens, JavaScript e fontes sejam acessados diretamente pelo navegador.

Melhora a performance

O Express lida de forma otimizada com arquivos estáticos, tornando o site mais rápido.

```
app.use(express.static(path.resolve(__dirname, 'views', 'public')))
```

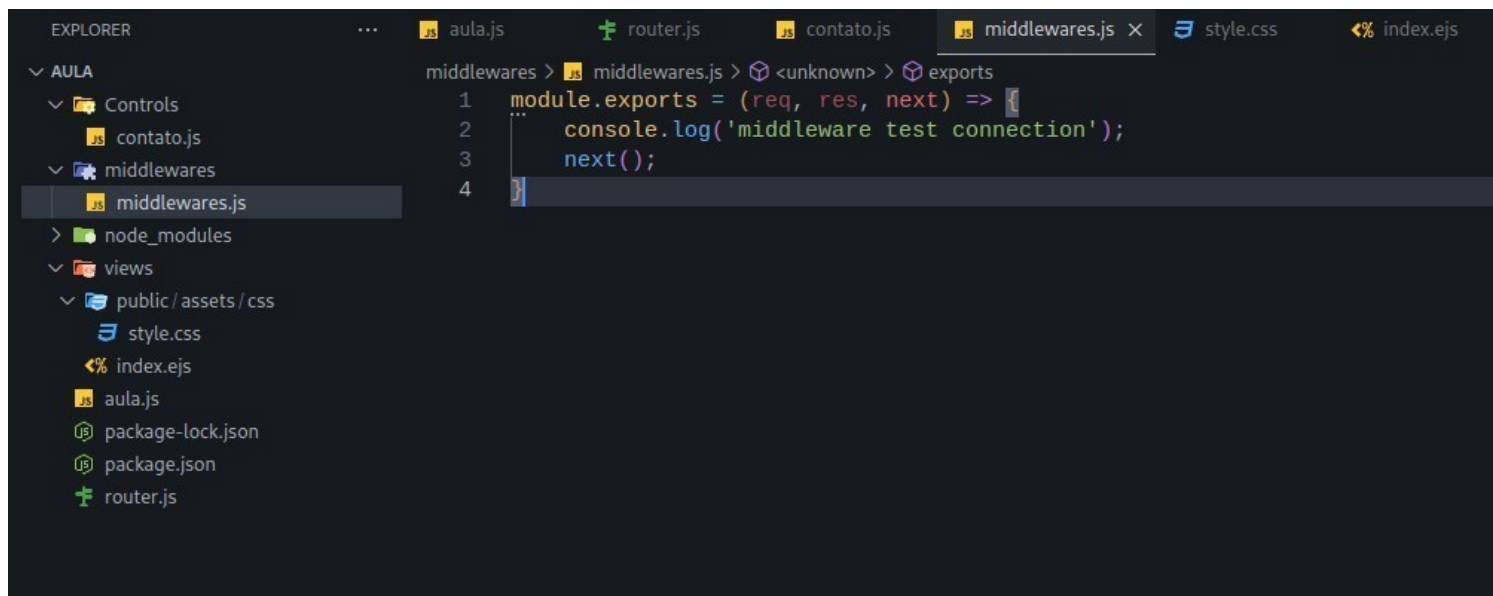


Caminho do arquivo

Middlewares

Fica entre a requisição do cliente e da resposta do servidor

Existe outro parâmetro depois de **req**, **res** que é o **next**(você pode usar o outro nome)
Ele basicamente vai fazer o middleware continuar



The screenshot shows the VS Code interface. On the left, the Explorer sidebar shows a project structure with folders like 'AULA', 'Controls', 'middlewares', 'node_modules', and 'views'. The 'middlewares' folder is expanded, showing 'middleware.js'. The main editor area shows the content of 'middleware.js' with the following code:

```
1 module.exports = (req, res, next) => {  
2   console.log('middleware test connection');  
3   next();  
4 }
```

Geralmente é criado uma pasta arquivos de middlewares



The screenshot shows the VS Code editor with the file 'aula.js' open. The code in the editor is as follows:

```
1 const express = require('express');  
2 const router = require('./router.js')  
3 const path = require('path');  
4 const app = express();  
5 const middleware = require('./middlewares/middleware.js')  
6  
7 app.use(express.urlencoded({extended: true}));  
8  
9 app.use(express.static(path.resolve(__dirname, 'views', 'public'))  
10  
11 app.set('views', path.resolve(__dirname, 'views'));  
12 app.set('view engine', 'ejs');  
13  
14 app.use(middleware);  
15 app.use(router);  
16  
17 app.use((req, res) =>{  
18   res.status(404).send('<h1>404 not found</h1>');  
19 });
```

O “.use” é para chamar alguma função middleware

Helmet e csrf

Vamos falar sobre segurança, principalmente contra falha XSS ou CSRF

Helmet é o mais fácil de usar, assim que você baixa o pacote(npm i helmet), você usa em como se fosse middleware, ele serve para que não seja injetado códigos no programa à partir do usuário.

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

```
const helmet = require('helmet'); #Sendo o primeiro middleware
```

Já o csrf deixa de ser simples, ele serve para que você tenha um token único. Onde você coloca em cada formulário. Vamos nos colocar em um cenário que o atacante consegue pegar o seu cookie para fazer autenticação em sua sessão, mesmo com o cookie em mãos, ele não irá conseguir, pois para fazer sessão, terá que ter um token, se o server ver que o token é diferente, ele logo recusa a conexão. Lembrando que ele fica armazenado na SESSION ou LOCALS. Agora vamos aprender a instalar em nosso programa:

```
const csrf = require('csrf');  
const {csrfMiddleware} = require('./middlewares/middleware');
```

Vamos importar

E em seguida vamos criar um middleware

```
middlewares > JS middleware.js > csrfMiddleware > csrfMiddleware  
1 exports.csrfMiddleware = (req, res, next) => {  
2   res.locals.csrfToken = req.csrfToken();  
3   next();  
4 }
```

```
app.use(helmet());  
app.use(refresh);  
app.use(csrf());  
app.use(csrfMiddleware);  
app.use(router);
```

Vamos usar nessa sequência
Iremos executar o csrf e usar nosso middleware para gerar os token's com a função .csrfToken() e colocar na variavel local

Na pasta controls e no seu arquivo que contem algum formulário ou campo para preencher, você irá exportar para o .ejs

```
exports.login = (req, res) => {  
  res.render('login.ejs', {csrfToken: res.locals.csrfToken});  
}
```

```
<div>  
  <form method="POST" action="/admin" class="form-class">  
    <input type="hidden" name="_csrf" value="<%= csrfToken %>">  
    <input type="text" name="user[email]" id="user_id" class="user-box-i<br>  
    <input type="password" name="user[pass]" id="user_pass" class="user-<br>  
    <button type="submit" class="btn-form">Enter</button>  
  </form>  
</div>
```

Ele tem que ser oculto(claro) então colocaremos 'hidden'

E em 'name' temos que colocar '_csrf'.. Quando o formulário é enviado, o middleware csrf verifica se o token enviado (_csrf) corresponde ao token armazenado na sessão ou no cookie. Se estiver correto, a requisição é processada. Se não, retorna um erro 403 Forbidden. Por fim, dentro do value colocamos a variavel local que contem o token