ex

ExpressJS

Instalando e executando

`npm i express`

```
express = require("express")

app = express()

app.use(express.json())

app.listen(5000, () => {console.log("servidor rodando")}
```

Outros downloads e observações

`npm i @types/express --save-dev`

`npm i nodemon --save-dev`

`npm i tsc --save-dev`

Rotas

Rotas são links específicos do nosso servidor os quais ao acessarmos, recebemos uma resposta.

Req e Res

Req e res são argumentos dos callbacks necessários nas funções do express.

`( req, res ) ⇒ {}`

`req` → Instância do objeto requisição

`res` → Instância do objeto resposta

Cada um possui métodos, e eles são os manipulados para enviar e receber mensagens.

Exemplo de get:

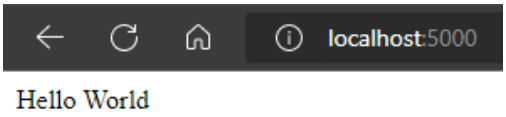
app.get()

`app. get ( ”rota” , ( req, res ) ⇒ {})`

Executa quando uma requisição get para a rota é feita, e faz tudo que está dentro da função callback.

O método `.send` de `res` envia um texto para o html no body do navegador.

```
app.get("/", (req, res) => {
  res.send("Hello World")
}
```



Parâmetros dinâmicos

Podemos especificar campos dinâmicos na nossa url, no sentido de, ao passar um valor arbitrário depois de um / ou usando ? parametros, podemos puxá-los dentro dos métodos.

`app.get(":id", (req, res) => {})`

O :id indica que, nessa rota, as coisas escritas depois do / vão ser guardadas em `req.params.id`

Se forem query parameters, ele ficará armazenado em `req.query.parametro`

Esse id é um nome arbitrário, pode ser chamado de qualquer coisa, mas podemos consultar o valor dentro da função.

É importante que rotas dinâmicas venham embaixo das outras rotas, pois uma rota com uma slug estática pode ser confundida com uma rota que recebe um valor dinâmico.

const router = express.Router()

A fim de organizar as rotas em arquivos separados, assim como componentes do React, pode-se separar a codificação de slugs específicas.

Exemplo:

Uma página que possui a seção users e a seção posts.

Pode se separar a configuração de rotas de users e de posts em dois arquivos diferentes, **users.js e seção.js.**

Dentro desses arquivos, deverá ser chamado um router, que faz o mesmo papel do app.

O export desse router é feito de forma diferente dos componentes do react, não sei por que.

> Em vez de importar com `import { express }` ou `import { nossarota }` usamos `require( arquivo/biblioteca )`.
>
> Para exportar, no final do código do módulo que queremos exportar:
>
> colocamos `module.export = variavel exportada`

```
const express = require("express")
const router  = express.Router()

router.get("/", (req, res) => {
  res.send("Oi!")
}
```

```
router.get("/example", (req, res) => {
  res.send("Example")
}

module.export = router
```

Ainda precisamos declarar uma slug que sirva como pai para essas rotas, nós fazemos isso ao importar o router dentro do main que possui o app.

```
const router = require("./users.js")
express = require("express")
app = express()

app.use("/users", router) // app.use("/{slug pai das rotas exportadas}", {roteador exportado})
app.listen(5000, () => {console.log("servidor rodando")}
```

Outros métodos

app.post()

`app.post( ”rota” , ( req, res ) ⇒ {})`

Para ler json de requests, usa-se o bodyparser no inicio do app

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

app.put()

`app.put( ”rota” , ( req, res ) ⇒ {})`

app.delete()

`app.delete( ”rota” , ( req, res ) ⇒ {})`

app.route()

O app.route é um método que, dada uma rota específica como /users, pode-se criar requisições de métodos http diferentes, sem ter que repetir a rota.

Exemplo:

```
app.route("users")
  .get((req, res) => {})
  .post((req, res) => {})
  .put((req, res) => {})
  .delete((req, res) => {})

// Ou

app.route("users").get((req, res) => {}).post((req, res) => {}).put((req, res) => {}).delete((req, res) => {})
```

Middleware

app.use(middleware) → Usa esse middleware em todas as requests.

Aulas Dave Gray

Padrões Regex nas rotas

Podemos inserir padrões regex na string das rotas para generalizar puxar certos padrões de link para certas rotas.

Redirecionamento

Dentro de uma rota, qualquer, para direcionar para outra usa-se `res.redirect("rota")`

É interessante mandar o status code de redirect (301 ou 302 nesse caso)

Rotas Inexistentes

Como o express lida com requests de cima pra baixo, podemos colocar uma request geral no fim do código para capturar páginas inexistentes.

É fato que o request geral também é ativado quando digitamos rotas que existem, mas como ele só é escrito no final, então não executa em rotas existentes pela existência das rotas de cima.

Cdigo

`app.method("/*"(req, res) => {})`

Esse request é ativado quando a rota não existe.

Route Handlers AKA Middleware

Route handlers são as funções callbacks chamadas dentro da chamada da rota.

```
app.get("/", (req, res) => {
  //Corpo da função
})
```

A parte verde da chamada é um route handler, é uma função executada quando a rota é requisitada.

É possível aninhar esses route handlers, e eles funcionam como uma espécie de middleware.

Para isso, é necessário que os handlers anteriores a outros possuam mais um argumento, **next**

Next é uma função que chama o próximo route handler

```
app.get("/", (req, res, next) => {
  //Corpo do handler 1
  next() -> Chama o próximo handler
```

```
}, (req, res) => {
  //Corpo do handler 2
})
```

Se esse next não for chamado, a próxima requisição não ocorrerá

Guardando handlers em variáveis

Handlers podem ser guardados em variáveis

```
const handler_1 = (req, res, next) => {
  //corpo
  next()
}

const handler_2 = (req, res, next) => {
  //corpo
  next()
}

const handler_3 = (req, res, next) => {
  //corpo
  next()
}
```

Esses handlers podem ser chamados em uma lista dentro da chamada da rota, fazendo os serem executados em sequência.

```
app.get("/", [handler1, handler2, handler3], (req, res) => {

})

// Ou apenas

app.get("/", handler1, (req, res) => {

})
```

A rota só será executada se houver a chamada do next no handler.

Pode-se criar aplicar funções assincronas dentro de um route handler para que ela faça operações desejadas.

App.use(middleware)

Alguns middlewares como `express.json()` ou `cors()`

Cors Boilerplate

```
const corsOptions = {
    origin: (origin, callback) => {
        if (whitelist.indexOf(origin) !== -1 || !origin) {
            callback(null, true)
```

```
        } else {
            callback(new Error('Not allowed by CORS'));
        }
    },
    optionsSuccessStatus: 200
}

// É interessante salvar essas opções em outra pasta
```

App.all(”rota”)

Se refere a todos os métodos http da request naquela rota.

MVC Model-View-Controller

Model View Controller é um design pattern de REST APIs,

Model se refere aos dados a serem entregues.

View é a página html.

Controller é o middleware que vai fazer as funções.

Define-se uma pasta de controllers com um crontroller específico para fazer mudanças no banco de dados e entregar ao frontend.

Controllers e middlewares

Um controller é um arquivo js com vários middlewares com `(req, res e/ou next)` que serão importados para um arquivo com o Router e este, por sua vez, ser importado na main.

JWT

Autenticação de JWT simples.

A autenticação deve ser feita em um controller, o loginController/authController

Ele é um middleware que recebe os dados de login e retorna um **AccessToken** e um **RefreshToken**

O **AccessToken vai ser armazenado em memória pela aplicação front end**, e será requisitado em todas as rotas.

Passo a passo:

O **RefreshToken será armazenado como um cookie httpOnly**

- Instalar a lib `“jsonwebtoken”`
- Ter certeza de que os body parsers de cookies e json estão ativos

```
import cookieParser from "cookie-parser"
//Também ter certeza que @types/cookie-parser está ativo

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

- Ter uma rota com um método POST que recebe as credenciais
  - Dentro dessa rota, verificar a senha do banco de dados e comparar com a informada (geralmente em hash)
  - Caso positivo, retornar um JWT
  - `import { sign } from "jsonwebtoken"`
  - Assinar o JWT, onde o payload é uma informação que você quer guardar.
    - Isso deverá ser feito para o AccessToken e para o RefreshToken
    - `const token = sign(payload, jwt_secret, opções_como_expiresIn)`
    - Isso retornará um token que poderá ser enviado ao client
  - Para o AccessToken, enviá-lo como JSON para o frontend `res.json({ accessToken })`
  - Para o RefreshToken, enviá-lo como um `res.cookie("nome", "valor", { httpOnly: true, maxAge: 24 * 60 * 60 * 1000 })`

Envio do accessToken

O accessToken recebido deve ser enviado por meio do Bearer, e será encontrado no header da requisição, no campo authorization.

`req.headers.authorization` → Retorna uma lista `["Bearer", Token]`

Troubleshooting criando apis

Ao armazenar hash de refresh token no banco de dados, a função hash do bcrypt faz com que, de alguma forma, a saída do hash fique aleatoria, então é meio impossível fazer uma comparação exata.

Esperado:

token → primeiroHash → hashedToken1

mesmoToken → segundoHash → hashedToken1

Acontecido:

token → primeiroHash → hashedToken1

mesmoToken → segundoHash → hashedToken2

Então, em vez de usar o bcrypt pra fazer hash, usei o crypto e a linha:

Erro ao dar deploy no heroku, erro de tsc.

Primeiro, precisa colocar alguns scripts especificos no node-modules:

```
"tsc": "tsc",
"build": "tsc",
```

```
  "start": "node index.js",
  "postinstall": "npm run build",
```

A segunda parte do erro, é desinstalar o tsc normal e reinstalar usando `npx --package typescript tsc --init`

Tutorial nesse link: This is not the tsc command you are looking for | bobbyhadz