

FIAP

FIAP

SLIDER ▢■◀



Next.js – API Handlers

O que é

API Handlers são uma funcionalidade do framework Next.js que permite criar APIs diretamente dentro da sua aplicação, sem a necessidade de configurar um servidor separado. Isso simplifica significativamente o desenvolvimento de aplicações full-stack, permitindo que você gerencie tanto a interface do usuário quanto a lógica do servidor em um único projeto.

Ao definir um **API Handler**, você cria uma rota específica para lidar com requisições HTTP, como GET, POST, PUT e DELETE. Essas rotas são separadas das rotas de suas páginas, permitindo que você organize melhor a sua aplicação.

Os **API Handlers** podem ser usados para uma variedade de tarefas, como buscar dados de um banco de dados, realizar autenticação, enviar e-mails, ou integrar sua aplicação com serviços de terceiros. A resposta de um **API Handler** é automaticamente serializada como JSON, facilitando o consumo por outras partes da sua aplicação ou por aplicações externas.

CRUD

CRUD é um acrônimo que representa as quatro operações básicas que podem ser realizadas em um banco de dados ou sistema de armazenamento de dados. Ele é fundamental para o desenvolvimento de software, especialmente em aplicações web e sistemas de gerenciamento de informações.

CRUD

Create (Criar)

Descrição: Refere-se à operação de criar ou adicionar novos registros em um banco de dados.

Exemplo: Adicionar um novo usuário a uma aplicação.

Read (Ler)

Descrição: Refere-se à operação de leitura ou recuperação de dados existentes de um banco de dados.

Exemplo: Buscar informações de um usuário específico a partir do banco de dados.

Update (Atualizar)

Descrição: Refere-se à operação de atualizar ou modificar registros existentes em um banco de dados.

Exemplo: Atualizar o endereço de um usuário no sistema.

Delete (Excluir)

Descrição: Refere-se à operação de excluir ou remover registros de um banco de dados.

Exemplo: Excluir um usuário do sistema.

Revisão HTTP

O **protocolo HTTP** define um conjunto de métodos que indicam a ação a ser realizada para um determinado recurso

GET

Solicita a representação de um recurso específico. As requisições GET devem apenas recuperar dados. Uso Comum: Carregar uma página da web, obter dados de uma API.

POST

Envia dados para o servidor criar um novo recurso. Os dados são incluídos no corpo da requisição. Uso Comum: Submissão de formulários, envio de dados de um aplicativo para o servidor

DELETE

Remove um recurso específico do servidor.

Uso Comum: Excluir uma conta de usuário, remover um item de uma lista.

Revisão HTTP

O **protocolo HTTP** define um conjunto de métodos que indicam a ação a ser realizada para um determinado recurso

PUT

Atualiza um recurso existente ou cria um novo se ele não existir, enviando os dados completos do recurso. Uso Comum: Atualizar informações de um usuário, substituir um recurso inteiro.

PATCH

Aplica modificações parciais a um recurso existente. Apenas os dados a serem atualizados são enviados. Uso Comum: Atualizar parcialmente um recurso, como modificar um único campo de um formulário.

Revisão HTTP - Headers

Os **headers** (cabeçalhos) de uma requisição HTTP são componentes essenciais que transmitem informações adicionais sobre a requisição ou a resposta. Eles fornecem metadados sobre a requisição, como o tipo de conteúdo, métodos de autenticação, e muito mais. Esses cabeçalhos ajudam o servidor a entender como processar a requisição e o cliente a interpretar a resposta.

Exemplos

Content-Type

Descrição: Indica o tipo de mídia do corpo da requisição.

Exemplo: Content-Type: application/json

Authorization

Descrição: Contém credenciais para autenticação da requisição.

Exemplo: Authorization: Bearer token

Fetch API

A **Fetch API** é uma interface moderna que permite fazer requisições HTTP de maneira fácil e flexível no JavaScript. Ela é baseada em Promises, o que facilita o tratamento assíncrono de operações.

- **url:** A URL do recurso que você deseja buscar.
- **options (opcional no GET):** Um objeto contendo qualquer configuração desejada, como método, cabeçalhos e corpo.

```
fetch(url, options)
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));
```

Fetch API

Exemplos

GET

```
fetch('https://api.exemplo.com/data')  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```

```
fetch('https://api.exemplo.com/dados', {  
  headers: {  
    'Authorization': 'Bearer MY_TOKEN',  
    'Accept': 'application/json'  
  }  
})  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```

Fetch API

Exemplos

POST

```
fetch('https://api.exemplo.com/dados', {  
  method: 'POST',  
  headers: {  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({ nome: 'João', idade: 30 })  
})  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```

Fetch API

Exemplos

PUT

```
fetch('https://api.exemplo.com/dados/1', {  
  method: 'PUT',  
  headers: {  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({ nome: 'João', idade: 30 })  
})  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error))
```

Fetch API

Exemplos

PATCH

```
fetch('https://api.exemplo.com/dados/1', {  
  method: 'PATCH',  
  headers: {  
    'Content-Type': 'application/json'  
  },  
  body: JSON.stringify({ idade: 32 })  
})  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```


Fetch API

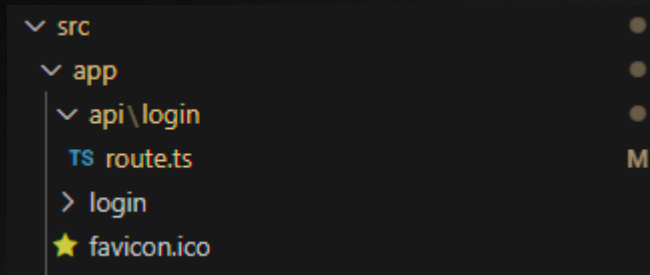
Exemplos

DELETE

```
fetch('https://api.exemplo.com/dados/1', {  
  method: 'DELETE'  
})  
  .then(response => response.json())  
  .then(data => console.log(data))  
  .catch(error => console.error('Error:', error));
```

Route Handler

1. Agora vamos criar efetivamente uma rota de login no back-end do Next. Ele tem o mesmo princípio do roteamento do front: Baseado em pastas e arquivos com nomes específicos. No caso do back-end o Next lê um arquivo com o nome de `route.ts`(ou `.js`) dentro de alguma pasta dentro do app. A princípio vamos criar uma rota de login e por organização vamos criar no seguinte path: `/api/login`, lá dentro vamos criar um arquivo `route.ts`



Route Handler

2. Feito isso vamos criar uma função Assíncrona com o nome de **POST**. É assim que o Next interpreta a criação das rotas através dos métodos, então se fossemos criar um GET, seria uma função com o nome GET. Lembrando que se precisarmos ter um POST e um GET (ou um DELETE e um PUT, tanto faz) dentro da mesma rota, devemos criar a declaração da função no mesmo arquivo.

Junto a isso já vamos criar os tipos da nossa função, tanto os de entrar (login e senha) quanto o

```
1
2 import { NextRequest, NextResponse } from "next/server"
3
4 // Interface de resposta da rota
5 type LoginResponse = {
6   token?: string
7   message?: string
8 }
9
10 // Interface do corpo da requisição
11 ...
12 interface LoginBody {
13   email: string
14   password: string
15 }
16
17 // Função que será executada quando a rota for chamada através do método POST
18 export async function POST(request: NextRequest): Promise<NextResponse<LoginResponse>> {
19
20 }
```

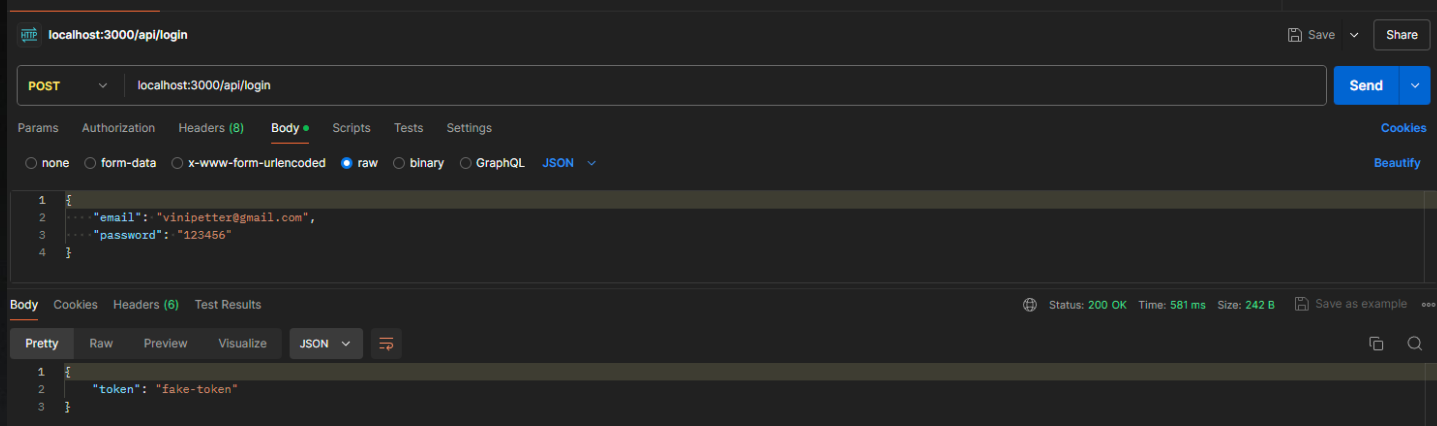
Route Handler

3. Vamos construir uma lógica provisória de login, apenas para ver se a nossa rota está funcionando corretamente. Mais pra frente vamos implementar corretamente

```
16 // Função que será executada quando a rota for chamada através do método POST
17 export async function POST(request: NextRequest): Promise<NextResponse<LoginResponse>> {
18   try {
19     // Lê o corpo da requisição
20     const body: LoginBody = await request.json()
21     const { email, password } = body
22
23     // Verifica se o email e senha foram informados
24     if (!email || !password) {
25       return NextResponse.json({ message: 'Email and password are required' }, { status: 400 })
26     }
27
28     // TODO: Implementar a lógica de autenticação
29     if (email === 'vinipetter@gmail.com' && password === '123456') {
30       return NextResponse.json({ token: 'fake-token' })
31     }
32
33     // Retorna erro de autenticação
34     return NextResponse.json({ message: 'Invalid email or password' }, { status: 401 })
35   } catch (error) {
36     console.error(error)
37     // Retorna erro interno do servidor
38     return NextResponse.json({ message: 'Internal server error' }, { status: 500 })
39   }
40 }
```

Route Handler

4. Agora para testarmos a nossa rota, vamos usar um client de chamadas, o POSTMAN (<https://www.postman.com/downloads/>). Lembrem de rodar o comando **npm run dev** E vamos fazer uma request do tipo POST para a nossa rota criada (**/api/login**) Podemos testar mandando email e senha errados, mandando vazio, e mandando o que está no nosso login de sucesso



Route Handler

5. Vamos criar dois arquivos simples para a nossa Dashboard logada, apenas para conseguir redirecionar o usuário para um tela após o login. Essa rota será acessada em /dashboard

```
src
├── app
│   ├── api\login
│   ├── TS route.ts
│   └── dashboard
│       ├── layouts.tsx
│       └── page.tsx
```

```
1 import { ReactNode } from "react"
2
3 You, 1 minute ago | 1 author (You)
4 interface DashboardLayoutProps {
5   children: ReactNode
6 }
7
8 const DashboardLayout = ({ children }: DashboardLayoutProps) => {
9   return <div className="flex flex-col gap-1 justify-center content-center h-full">
10     <header className="text-center px-2">
11       <h1 className="text-xl">
12         Dashboard
13       </h1>
14     </header>
15     <main className="px-2">
16       {children}
17     </main>
18   </div>
19 }
20 export default DashboardLayout
```

```
rc > app > dashboard > page.tsx > default
1 const HomeDashboardPage = () => {
2   return <>
3     <h6 className="text-center" style={{ minHeight: 'calc(100vh - 36px)' }}>
4       Home Dashboard Page
5     </h6>
6   </>
7 }
8
9 export default HomeDashboardPage
```

Route Handler

6. Agora vamos voltar para a página de login e usar o fetch para fazer o login na rota que criamos. Vamos o fetch para bater em /api/login, usar o metodo POST e passar o login e a senha no body da nossa request.

Vamos pegar a resposta a validar se voltou um token, caso contrario lançar um erro. Em caso de sucesso vamos redirecionar para a página /dashboard.

Além disso precisamos importar o router do Next

```
async function submitCallback(values: FormState) {
  try {
    // Enviar requisição para a API
    const request = await fetch('/api/login', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(values)
    })
    // Ler a resposta da API
    const response = await request.json()

    // Verificar se a resposta contém um token
    if (!response.token) {
      throw new Error(response.message)
    }

    // TODO: Guardar Token

    // Redirecionar para a página de dashboard
    router.push('/dashboard')
  } catch (error) {
    // Tratar erro
    if (error instanceof Error) {
      return submitErrorCallback(error)
    }
    return submitErrorCallback(new Error('Erro ao realizar login'))
  }
}
```

Route Handler

7. Além disso precisamos importar o router do Next, que estamos usando no final da nossa função de submit

```
import Button from "@components/button/button"
import Input from "@components/input/input"
import useForm, { FormState } from "@hooks/use-form/use-form"
import { useRouter } from "next/navigation"
import { useRef } from 'react'

const LoginForm = () => {
  const router = useRouter()
```

Route Handler

8. Agora vamos melhorar nossa função de erro, para exibir os erros corretamente.

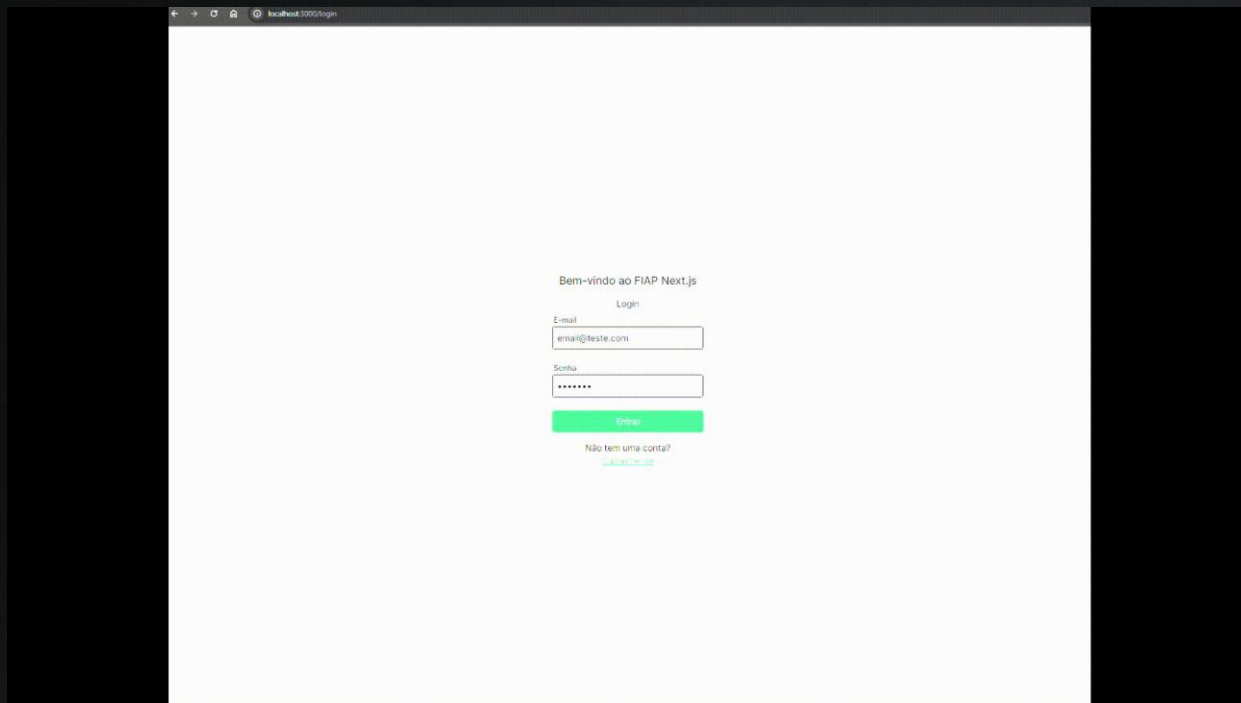
Vamos validar se o nosso objeto de erro tem uma propriedade chamada `cause` e se elas possuem chaves validas. Se tiver vamos montar uma mensagem com todas as causas, se não vamos exibir o message do erro diretamente em um alert mesmo. (Você pode trabalhar melhor a forma de mostrar um erro, como mostrando em um modal ou em algum lugar do formulário)

```
async function submitErrorCallback(error: Error) {  
  // Verificar se o erro contém causas  
  if (error.cause && Object.keys(error.cause).length) {  
    // Mostrar mensagem de erro para cada causa  
    let message = 'Erro ao realizar login:\n\n'  
    for (const key in error.cause) {  
      // Adicionar causa ao texto da mensagem  
      const causes = error.cause as { [key: string]: string }  
      message += `- ${causes[key]}\n`  
    }  
    // Exibir mensagem de erro  
    return window.alert(message)  
  }  
  
  // Exibir mensagem de erro quando não houver causas  
  return window.alert(error.message)  
}
```



Route Handler

Resultado



A screenshot of a web browser window displaying a login page. The browser's address bar shows 'localhost:3000/login'. The page content is centered and includes the following elements:

- Greeting: Bem-vindo ao FIAP Next.js
- Form Title: Login
- E-mail field: A text input containing 'email@teste.com'.
- Senha field: A password input with masked characters '*****'.
- Login Button: A green button labeled 'Enviar'.
- Registration Link: A link labeled 'Não tem uma conta?' with a green underline.

Exercício

Faça como eu fiz e crie um Route Handler para a rota de cadastro para a tela que foi criada nas aulas anteriores. O cadastro deve ser um POST para `/login/register`. A princípio como não temos banco de dados, podemos só exibir uma mensagem de sucesso e redirecionar o usuário para a tela de login (ou dashboard, como preferir)

Dúvidas, críticas ou sugestões?

FIAP