4 - Renderização do Lado do Servidor (SSR) no Next.js

E aí, programador! o/

Nessa aula vamos ver na prática como trabalhar com a renderização do lado do servidor no Next.js.

1. Antes de começar, vamos instalar o bootstrap e o reactstrap para nos ajudar com a estilização das nossas páginas, assim podemos ter uma página mais bonita sem perdermos o foco da série que é o Next.js

```
npm i bootstrap reactstrap
```

2. Agora podemos excluir os estilos padrão que vieram na instalação e incluir o css do bootstrap alterando o arquivo _app.tsx:

   Obs.: Esse arquivo é o que envolve toda a nossa aplicação. Se quisermos adicionar algo globalmente podemos utilizar esse arquivo.

```
// pages/_app.tsx

import 'bootstrap/dist/css/bootstrap.min.css'
import type { AppProps } from 'next/app'

function MyApp({ Component, pageProps }: AppProps) {
  return <Component {...pageProps} />
}

export default MyApp
```

3. Agora vamos criar uma página para a nossa demonstração chamada dynamic.tsx.

   Obs.: Criaremos páginas separadas para cada explicação pois o uso da renderização no servidor exclui a possibilidade de gerar uma página estática e vice-versa, então teremos uma página para cada exemplo.

```
// pages/dynamic.tsx

import { NextPage } from "next"
import { Col, Container, Row } from "reactstrap"

const Dynamic: NextPage = () => {
  return (
    <Container tag="main">
      <h1 className="my-5">
        Como funcionam as renderizações do Next.js
      </h1>

      <Row>
        <Col>
          <h3>
            Gerado no servidor:
          </h3>
        </Col>
```

```
        <Col>
          <h3>
            Gerado no cliente:
          </h3>
        </Col>
      </Row>
    </Container>
  )
}

export default Dynamic
```

4. Já temos a página disponível. Vamos agora alterar levemente a rota de API “hello” que veio na aplicação. Vamos fazer ela incluir uma informação um pouco mais dinâmica para simularmos dados de um back-end que mudam constantemente. Para isso vamos incluir uma timestamp na resposta que retorna a data e hora atuais:

   Obs.: Teste no próprio navegador ou em programas como Postman e Insomnia para ver que a API retorna um valor atualizado a cada chamada.

```
// pages/api/hello.ts

// Next.js API route support: https://nextjs.org/docs/api-routes/introduction
import type { NextApiRequest, NextApiResponse } from 'next'

type Data = {
  name: string
  timestamp: Date
}

export default function handler(
  req: NextApiRequest,
  res: NextApiResponse<Data>
) {
  const timestamp = new Date()
  res.status(200).json({ name: 'John Doe', timestamp })
}
```

5. Agora podemos comparar as renderizações do lado do cliente e do servidor para vermos as suas diferenças. Primeiro vamos incluir uma chamada a API como faríamos em uma aplicação React comum, ou seja, usando os hooks. No arquivo dynamic.tsx adiciona o seguinte código:

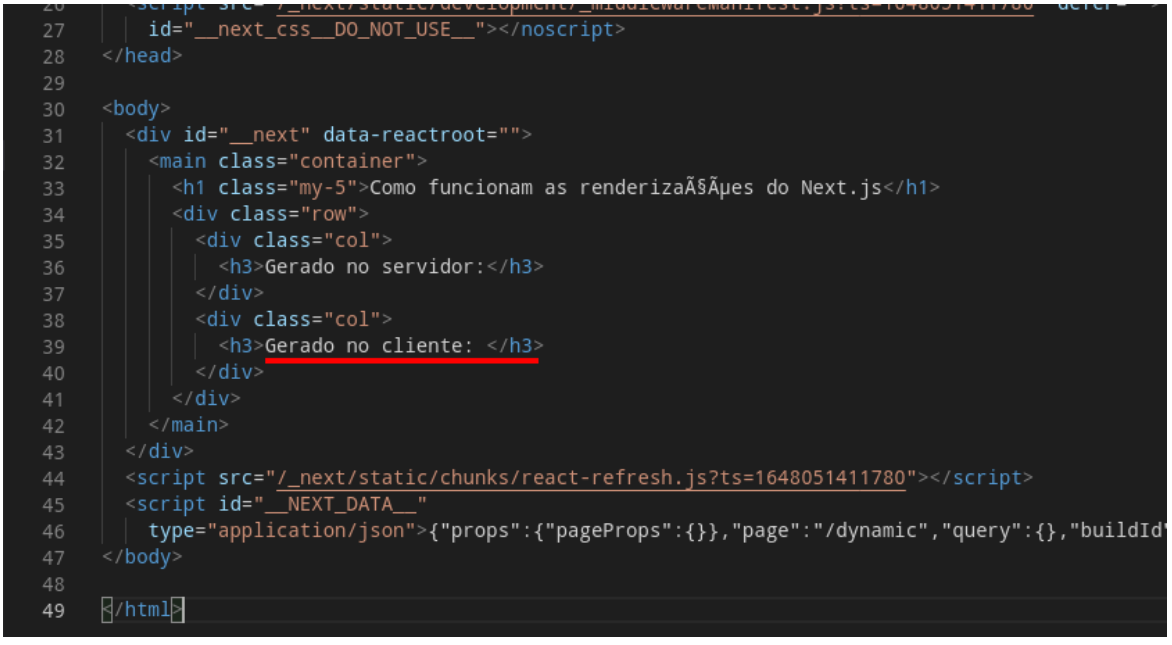
   Obs.: Se testarmos a página /dynamic agora veremos que a cada atualização ela obtém uma timestamp atualizada.

```
// pages/dynamic.tsx

import { NextPage } from "next"
import { useEffect, useState } from "react"
import { Col, Container, Row } from "reactstrap"

type ApiResponse = {
  name: string
  timestamp: Date
}

const Dynamic: NextPage = () => {
  const [clientSideData, setClientSideData] = useState<ApiResponse>()

  useEffect(() => {
    fetchData()
  }, [])
```

```
  const fetchData = async () => {
    const data = await fetch("/api/hello").then(res => res.json())
    setClientSideData(data)
  }

  return (
    <Container tag="main">
      <h1 className="my-5">
        Como funcionam as renderizações do Next.js
      </h1>

      <Row>
        <Col>
          <h3>
            Gerado no servidor:
          </h3>
        </Col>

        <Col>
          <h3>
            Gerado no cliente: {clientSideData?.timestamp}
          </h3>
        </Col>
      </Row>
    </Container>
  )
}

export default Dynamic
```

6. Porém esse método, como eu disse antes, tem um problema de SEO grave. Utilize a função de ver o código-fonte da página no seu navegador, cole esse código fonte no VS Code, formate o código e veja que o HTML da página não inclui os dados dinâmicos que obtivemos. Agora imagine isso em todos os posts de um blog ou todos os produtos de uma loja obtidos dinamicamente? O site aparentaria estar vazio, o que machucaria o SEO das páginas.

   Obs.: Vale destacar aqui também que o Next.js otimiza até mesmo o nosso html, por isso ele aparece de forma tão ilegível no navegador.



7. Agora vamos ver uma possível solução para esse problema, a função getServerSideProps. Ao utilizá-la estaremos indicando que a página em questão contém dados dinâmicos que devem estar no HTML, portanto o Next.js irá renderizar previamente o HTML no lado do servidor com todos os dados. Vamos chamar a mesma API mas dessa vez com o getServerSideProps:

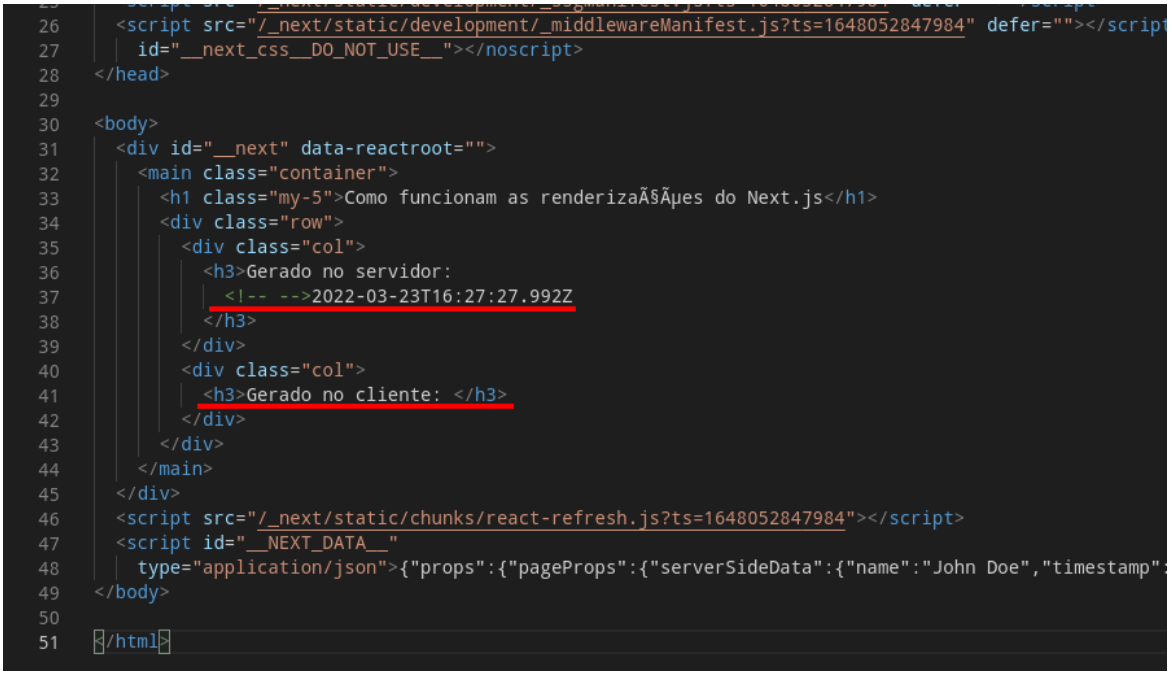
```
// pages/dynamic.tsx

import { GetServerSideProps, NextPage } from "next"

// ...

export const getServerSideProps: GetServerSideProps = async () => {
  const serverSideData: ApiResponse = await fetch(`${process.env.NEXT_PUBLIC_APIURL}/api/hello`).then(res => res.json())

  return {
    props: {
      serverSideData
    }
  }
}

const Dynamic: NextPage = (props: {
  children?: ReactNode
  serverSideData?: ApiResponse
}) => {
  const [clientSideData, setClientSideData] = useState<ApiResponse>()

// ...

  return (
    <Container tag="main">
      <h1 className="my-5">
        Como funcionam as renderizações do Next.js
      </h1>

      <Row>
        <Col>
          <h3>
            Gerado no servidor: {props.serverSideData?.timestamp}
          </h3>
        </Col>

        <Col>
          <h3>
            Gerado no cliente: {clientSideData?.timestamp}
          </h3>
        </Col>
      </Row>
    </Container>
  )
}

export default Dynamic
```

8. Repare que dessa vez estamos utilizando uma variável de ambiente para a url da API. Isso é porque nesse caso o Next.js só aceita URLs absolutas. Para incluir as variáveis de ambiente durante o desenvolvimento basta criar um arquivo “.env.development.local” na raiz do projeto. Ele já está sendo ignorado pelo .gitignore:

   Obs.: Repare que ao invés do tradicional REACT_APP_ usado com o create-react-app nós temos NEXT_PUBLIC_ para variáveis de ambiente disponíveis publicamente na aplicação. Como nossa aplicação também roda do lado do servidor isso permitiria termos variáveis de ambiente que não ficam públicas. Outra vantagem do Next.js.

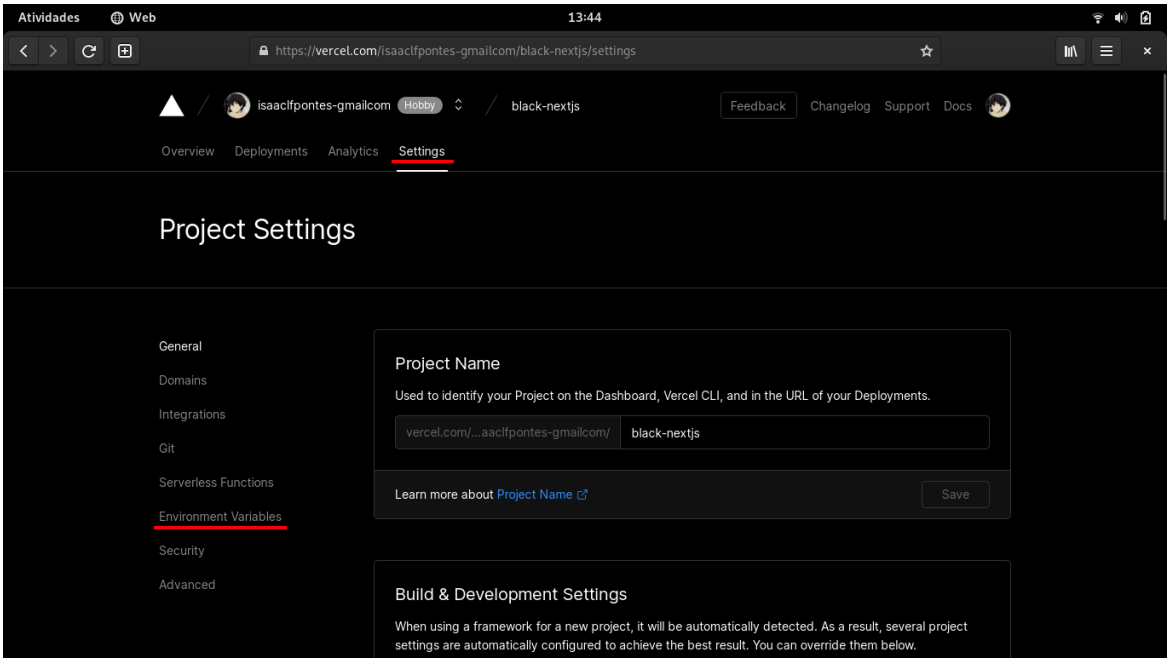
```
// .env.development.local

NEXT_PUBLIC_APIURL=http://localhost:3000
```

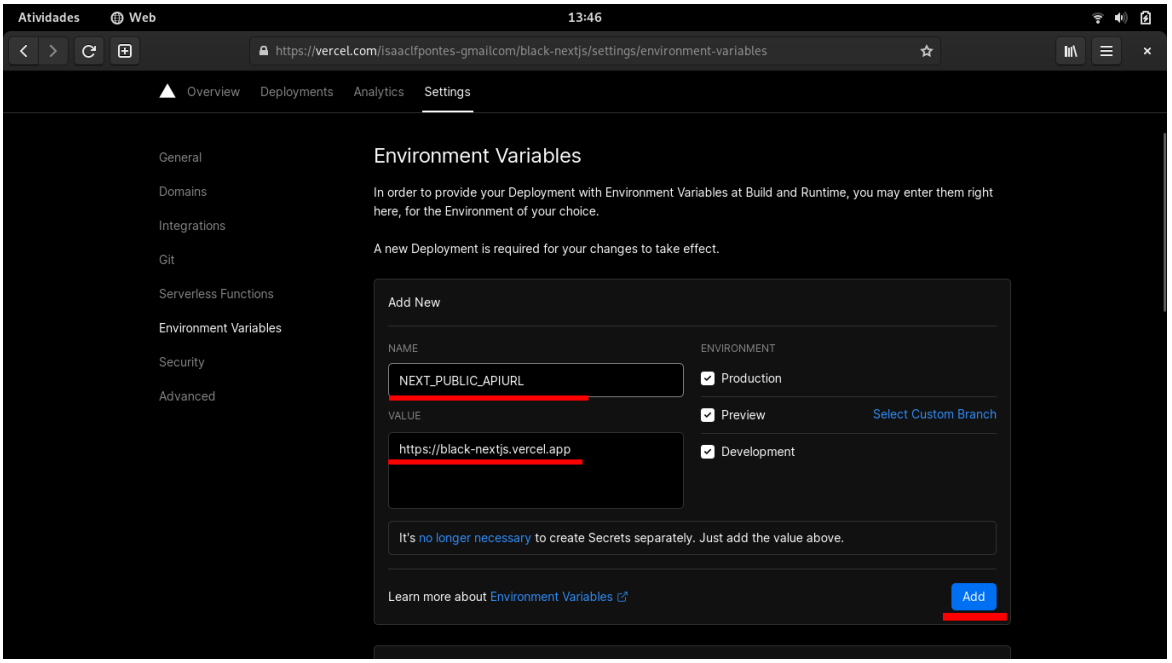
9. Como as variáveis de ambiente foram modificadas, reinicie a aplicação e teste a página. Agora podemos ver o resultado lado a lado no código-fonte:



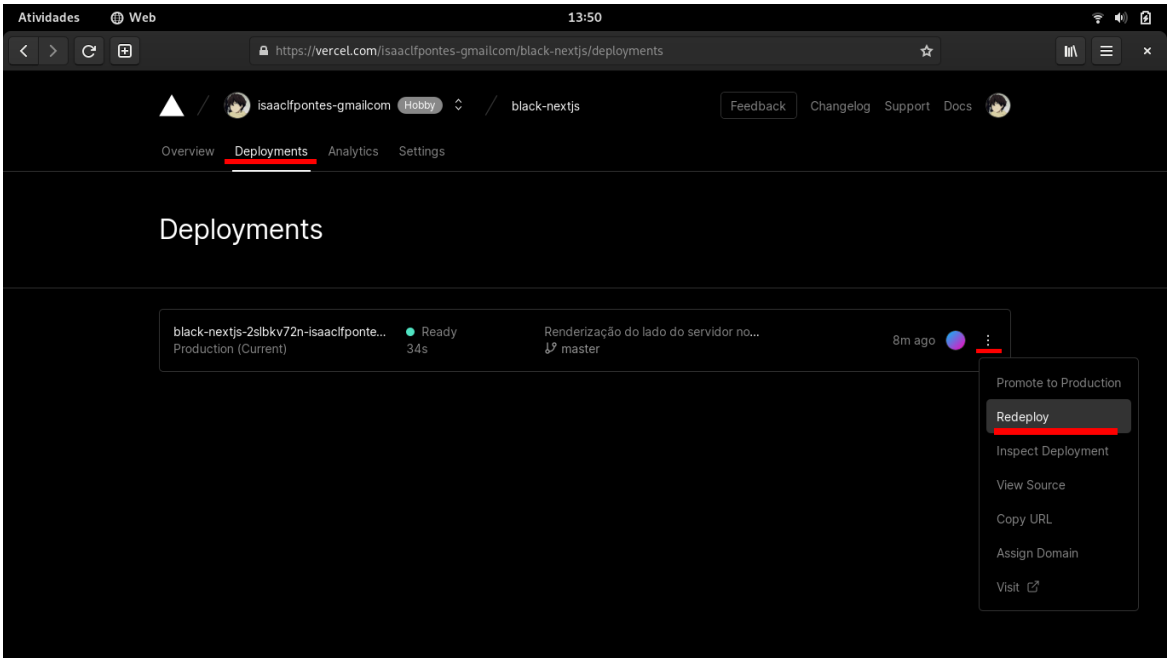
10. Antes de fazer o commit e push da nossa aplicação e ver as mudanças refletidas na versão de produção existe uma parte que precisamos fazer manualmente, que é a inclusão da variável de ambiente da nossa URL. Para incluir a variável siga os passos abaixo no seu painel da Vercel:

    Obs.: Não se esqueça de utilizar a URL do seu app no valor da variável.





11. Agora podemos fazer o commit e o push. Caso você já os tenha feito antes terá que realizar o um novo deploy para ver as mudanças nas variáveis de ambiente ter efeito:


Atividades
Web
13:50
https://vercel.com/isaaclfpontes-gmailcom/black-nextjs/deployments
isaaclfpontes-gmailcom
Hobby
black-nextjs
Feedback
Changelog
Support
Docs
Overview
Deployments
Analytics
Settings
Deployments
black-nextjs-2slbkv72n-isaaclfponte...
Production (Current)
Ready
34s
Renderização do lado do servidor no...
master
8m ago
Promote to Production
Redeploy
Inspect Deployment
View Source
Copy URL
Assign Domain
Visit