JUNIORDev
DIGITALNA DALMACIJA

JSX

Prije nego nastavimo sa primjerima, kratko ćemo se osvrnuti na sintaksu u prethodnom primjeru. React koristi JSX (JavaScript + XML) koji na prvi pogled sliči "klasičnom" HTML-u, ali nam dozvoljava pisanje vlastitih oznaka i umetanje JS izraza.

Sav kôd koji je napisan u JSX-u se na kraju prevodi u JavaScript - zbog toga nam je i potreban lokalni razvojni poslužitelj koji može odmah naše promjene prikazati na ekranu preglednika.

Na primjeru komponente iz prethodnog poglavlja možete vidjeti kako ista komponenta izgleda u JSX obliku, te kako u konačnom prevedenom (JavaScript) obliku:

JSX JavaScript

index.js

```
const App = () => {
  const datum = new Date()
  const a = 10
  const b = 20

  return (
    <div>
      <p>Dobar dan React, danas je {datum.toString()}</p>
      <p>
        {a} plus {b} je {a + b}
      </p>
    </div>
  )
}
```

Tehnički gledano, nije potrebno koristiti JSX već React aplikacije možete pisati u "čistom" JavaScriptu ali tako nešto nije naročito praktično.

Prevođenje

Prevođenje iz JSX u JS se odvija uz pomoć prevoditelja ili kompajlera (*compiler*) pod imenom "**Babel**". Babel nam služi kako bi se postigla kompatibilnost sa prethodnim verzijama JS-a. Aplikacije napravljene pomoću *create-react-app* skriptom ili *Vite* alatom imaju automatski postavljen i konfiguriran Babel. U slučaju kada bi samostalno izrađivali vlastitu React aplikaciju, morali bi u sklopu projekta instalirati i konfigurirati Babel ili neki drugi prevoditelj po želji. Detaljnije o samom Babel-u možete pogledati na službenoj stranici ali za potrebe ovog tečaja nećemo ulaziti u prevođenje JavaScripta.

Pakiranje

Još jedan koncept koji je korisno spomenuti je pojam *bundler-a*. Radi se o alatima koji pomažu u upravljanju i organizaciji JavaScript kôda, čime se olakšava rad sa velikim i složenim aplikacijama. Za vrijeme razvoja aplikacije, programerima je preglednije pisanje programa u odvojenim datotekama koje su složene u nekoj hijerarhiji. Bundler programi uzimaju sav vaš kôd i sve potrebne (*dependency*) datoteke te ih pakiraju u jednu optimiziranu datoteku. Ovo olakšava učitavanje vašeg koda, smanjuje broj zahtjeva prema poslužitelju i može pomoći u poboljšanju performansi vaše aplikacije. Neki od najpopularnijih JavaScript bundlera su Webpack, Parcel i Rollup. Vite aplikacija koju koristimo u ovim primjerima trenutno koristi "Rollup", ali to je još jedna od stvari koje su odrađene u pozadini umjesto nas i za početak se ne moramo brinuti o tome.

JSX sintaksa

Vratimo se na JSX sintaksu. Kao što smo vidjeli u prethodnim primjerima JSX nam omogućava "kombiniranje" HTML i JS kôda tj. dodavanje dinamičkog sadržaja unutar HTML strukture. Dinamički sadržaj se umeće unutar "" zagrada i evaluira prilikom prikazivanja komponente (kao što smo vidjeli u primjeru sa prikazom datuma). Iako je JSX na prvi pogled sličniji HTML-u (barem kod jednostavnijih komponenti, jer većina kôda otpada na strukturu), zapravo se radi o

nadogradnji JavaScript sintakse. Sintaksa JavaScript-a je malo "stroža" od HTML-a, pa to isto vrijedi i za JSX. Zbog toga prilikom pisanja JSX-a moramo voditi računa o nekoliko osnovnih pravila:

1. Jedan element

Funkcija koja predstavlja komponentu kao povratnu vrijednost vraća izgled komponente. Pri tome, *return* izraz mora uvijek vraćati samo **jedan** *root* element. Naravno, struktura komponente je uglavnom sastavljena od više HTML elemenata, ali oni moraju biti ugniježđeni unutar jednog glavnog elementa. Ukoliko imate više elemenata koji su logički na istoj razini, najjednostavnije rješenje je sve ugnijezditi unutar jednog praznog *div* elementa:

```
function Primjer(){
  return(
    <div>
      <h1>Naslov komponente</h1>
      <p>Ovi elementi su na istoj razini</p>
      <p>pa koristimo div kao root element</p>
    </div>
  )
}
```

U slučaju da ne želite dodatno kompliciratи strukturu svoje stranice (jednom kada se prevede u "čisti" HTML), možete jednostavno koristiti prazne oznake:

```
function Primjer(){
  return(
    <>
      <h1>Naslov komponente</h1>
      <p>Ovi elementi su na istoj razini</p>
      <p>pa koristimo praznu oznaku kao root element</p>
    </>
  )
}
```

2. Zatvaranje svih oznaka

Jedno od pravila JSX sintakse je da sve oznake moraju biti eksplicitno zatvorene. HTML je "popustljiviji" po tom pitanju i sadrži tzv. samozatvarajuće oznake (kao npr `<img>`) kojima možete ali ne morate staviti znak `"/"` na kraju. Kod JSX-a nemamo tu slobodu, već sve oznake moraju imati strogo definiran kraj.

```
<img src='./slika.jpg' alt='primjer' />
```

3. *camelCase* atributi

Kôd napisan u JSX-u se na kraju prevodi u JavaScript i atributi koje navodimo uz oznake postaju ključevi (*key*) u JS objektu. JavaScript sadrži neka ograničenja pri imenovanju ključeva i varijabli. Već smo vidjeli primjer sa ključnom riječi *class* koja je rezervirana za definiranje klasa, pa moramo koristiti varijantu *className* kako bi HTML oznakama dodali atribut klase. JS također ne dozvoljava ni crtice u imenima pa atribute kao npr *stroke-width* pišemo u obliku ***strokeWidth***.

Ovaj oblik pisanja imena se naziva *camelCase* i karakterističan je po tome što se riječi pišu bez razmaka (tj. bez crtica) i svaka nova riječ započinje velikim slovom (osim prve riječi). Ovo pravilo vrijedi i za atribute koji nisu napisani u tom stilu npr. ***onclick*** zbog toga u JSX-u pišemo kao ***onClick***.

Postoji jako malo iznimki od toga pravila - atributi `aria-*` i `data-*` se pišu identično kao i u HTML-u. Na prvu se ovo sve čini komplicirano ali ukoliko krivo napišete neki atribut, React prevoditelj će ispisati pogrešku i upozoriti vas na krivo korištenje sintakse. Također, uvijek možete pogledati popis događaja i svojstava u službenoj dokumentaciji, gdje su navedena ispravna imena i način korištenja.

U idućem poglavlju ćemo se vratiti na praktične primjere i detaljnije ćemo se osvrnuti na React komponente.

Last updated on March 7, 2023

Uvod

Komponente