Dokumentation

Inledning

Vi har delat upp filerna och numrerat dem för Views, Sections, Assets, Components och Contexts. I Views har vi home, features och contact, som använder BrowserRouter för att navigera mellan sidorna trots att det är en single page-applikation.

Från Sections-mappen har vi delat upp det i mappar relaterade till sina respektive routes. Home-routen har sina sektioner, och contact har sina. Vi har sedan använt imports i Views för att placera våra komponenter i en fragment.

Home-sidan innehåller sina vanliga sektioner från HTML/CSS-kursen men har nu ett extra fokus på Reviews-sektionen, Q&A-sektionen och Subscribe-sektionen längst ned.

Reviews sektionen

Reviews-sektionen har en useState som håller en array. Vi har en fetchData-funktion som asynkront hämtar data från ett API med fetch-funktionen. Fetch returnerar ett promise som vi sparar i response, och sedan konverterar vi till JSON i dataInJson. Vi har detta i en try/catch-block för att kunna hantera eventuella felmeddelanden. Denna funktion körs i bakgrunden utan att

orsaka state-rendering tack vare useEffect. Vi returnerar sedan diven vi tidigare hade, men nu mappar vi istället en ReviewBox-komponent där vi skickar ned värdena från JSON-objektet vi fick från fetchen. Vi skickar ned text, rating, avatarUrl, author och jobRole.

Reviews Box komponenten

Inuti ReviewBox-komponenten använder vi PropTypes-validering och hämtar bilderna för FilledStar och UnfilledStar. Vi har sedan två const-variabler där vi tar propen rating och fyller arrayen med filled stars baserat på ratingen. Därefter tar vi unfilledStar-arrayen och fyller den med en total siffra på 5 minus ratingen, så att den ger de återstående stjärnorna upp till 5. Inuti star-box lägger vi sedan dessa två arrays bredvid varandra, vilket ger totalt 5 visade stjärnor. Därefter lägger vi till text, namn, img, username och title på rätt ställen eftersom de mappades i Review-sektionen.

Q & A Sektionen

QA-sektionen, eller FAQ-sektionen som det egentligen är, har vi tagit hela sektionen och konverterat till JSX. Därefter har vi skapat en Accordion-komponent. Accordion-komponenten fungerar så att den har två state-hanterare och en useEffect som körs i bakgrunden för att hämta data från API
via FAQ-URL
. Den väntar in responsen, konverterar den och placerar den i qaItems state-hanteraren. Vi mappar ut qaItems som

JSX-element där vi skickar med item och index från map-funktionen. Vi använder toggleAccordion vid klick, som använder index för att antingen öppna eller stänga genom att ändra state för activeIndex till null eller till index, beroende på vad index är. Accordion-CSS-selektorn läggs på eller tas av baserat på state.

Subscribe Form

Sedan har vi på homepage en sektion som kallas Contact men som innehåller komponenten SubscribeForm. Här har vi laddat ner dependencies för ramverken Formik och Yup, som är en formulärhanterare och validerare. Med hjälp av Yup skapar vi ett valideringsschema som vi kopplar ihop med fälten i Formik. Vi returnerar JSX med en komponent som heter Formik, som tar initialvärden, ett valideringsschema och en onSubmit-funktion. Inuti detta ramverk använder man Field som fält för inputs, och när man namnger dem med name kopplas de till valideringsschemat. Detta schema validerar då automatiskt på onBlur, onSubmit, etc. för att visa felmeddelanden. Via Formik använder vi isSubmitting, errors och touched för att visa felmeddelanden under input-fältet. När formuläret skickas in körs handleSubmit, vilket skickar en POST-request till API som returnerar information i konsolen.

Features vy

Inuti denna komponent gör vi inget särskilt förutom att återanvända Appfeatures- och HowItWorks-komponenterna, vilket visar fördelen med att kunna återanvända komponenter i React med bara en eller några rader kod.

Contact

Inuti kontaktvyn har vi delat upp det i sektionerna contact hero och contact map. Kontaktsidan autogenererades via builder.io, och vi modifierade de divar och texter som behövdes för att få sidan responsiv och för att lösa eventuella buggar.
Hero-sektionen har sin text placerad och komponenten med kontaktformuläret. Här valde jag att använda en återanvändbar komponent, så jag delade upp komponenten i ett start- och sluttag och skickade sedan in children-komponenter. Dessa komponenter placeras individuellt mellan start- och sluttaggen med de attribut som krävs för varje komponent, såsom namn, options osv. De komponenter som skickas ned är Label, inputfält och en select. Dessa komponenter använder field för att kunna hämta ErrorMessage från Formik.

Själva kontaktformuläret har en errorMessage-state, ett valideringsschema från Yup, initialvärden, och en handleSubmit som postar data och returnerar information. Formulärets JSX tar komponentens children-props och placerar dem under h2-titeln för varje komponent vi angett. Detta gör det möjligt att återanvända ContactForm och skicka ned andra fält, eller samma fält med andra namn. Dock skulle vi behöva lyfta upp valideringsschemat och skicka det som en prop eller i en context för att kunna hantera olika valideringar i varje ContactForm.

Contact Map

Här har vi börjat använda useContext. I Contexts-mappen har vi en komponent som heter LocationContext. Här skapar vi en context med hjälp av createContext och namnger den LocationContext. Vi har ett state med lat, lng och namn, samt en useEffect för att hämta data från ett API som Google tillhandahåller. Man behöver skapa ett konto och generera en API-nyckel. Vi har sedan fetchLocationName som hämtar Google API

tillsammans med navigator.geolocation.getCurrentPosition och skickar in latitude och longitude i fetch-funktionen.

Vi returnerar JSX som en komponent där vi skickar med location från state och children, som vi anger i appen. Viktigast är att vi exporterar contexten, vilket ger oss möjlighet att modifiera och ändra contexten från alla children inuti LocationProvider.

Contact Map Section

Här använder vi contexten vi skapade tidigare. Vi destrukturerar contexten från useLocation() och namnger lat, lng och name som initiala värden, eftersom de redan har sina värden från den nuvarande platsen. När komponenten laddas med useEffect placeras initialvärdena för state för lng/lat/name. Inuti JSX har vi en ContactMap-komponent som tar emot dessa lat/lng och namn som props.

Innan vi går vidare till den komponenten har vi två funktioner, handleAdressClick och resetAdressBack, som använder state för att ange nya värden som vi har kopplat till p-elementen med adresserna. Koordinaterna är hårdkodade. En reset-knapp skapades för att återställa state till initialvärdena. ContactMap-komponenten består av en props som tar emot lat, lng och name, samt en GoogleMap-komponent som tar emot styles, center, zoom och marker-position som anges med de state-props som skickats in.

Top-Header

Slutligen har vi den sista komponenten, placerad i app.jsx utanför Routes och main för att alltid vara renderad oavsett route. Vi har lagt denna i Sections under universal sections. Här använder vi Link och NavLink från react-router-dom, vilket gör att vi kan navigera mellan rutterna. Vi har även en DarkModeSwitch-komponent och en HamburgerMenu-komponent som tar emot en toggleMenu-funktion som växlar true/false på isMenuOpen-statet. Menyknappen tar bara emot och kör funktionen.

Längst ned shortcurtar vi en meny om isMenuOpen är true, med komponenten HiddenMenu. Där använder vi useRef som fungerar som en variabel som bibehålls mellan states. Vi sätter menuRef till nav-elementet och checkar onClick på länkarna samt onBlur på naven. OnBlur anropar en funktion, handleBlur, där vi kontrollerar om menuRef.current (naven) inte innehåller eventets relatedTarget, och då stänger vi menyn. Vi har även en useEffect

som körs med en eventListener för mousedown, och så länge menuRef är aktiv och inte har samma target som menuRef eller Selector hamburger-menu, så stängs menyn.