Rapport – **A Minimum Viable Product**

Länk till kod på GitHub: https://github.com/NikBjo72/eyemarketing

Min återanvändningsbara komponent

Jag har två återanvändningsbara komponenter (blinking-eye-btn.jsx och collapsable-fieldset.jsx) dock har jag fokuserat på blinking-eye-btn.jsx.

Syftet med komponenten är att på ett enkelt sätt återanvända en knapp som byter bild när man klickar på den samt utför en uppgift. Helt enkelt slippa skriva samma kod en gång till i applikationen. Nu behöver man bara ange olika props för att den ska kunna återanvändas i olika sammanhang.

Knappen ska kunna användas som navigeringsknapp i en meny (typ=”global”), där den byter mellan sidor/komponenter, känner av vilken sida som är aktiv och då justerar tillbaka till grundbilden på de knappar som inte är aktiva. För just denna typen av knapp (global) behöver React Router vara installerat, då jag använder useLocation från react-router.

Det andra sammanhanget jag också vill att den ska kunna användas i, är som en ”toggle”-knapp, där den visar något när den är aktiv, och tar bort när den är inaktiv(type=”local”).

Det tredje alternativet (type=”local-scope”), är när den används i t.ex. en egen komponent, med flera knappar, där knapparna ”togglar”, men bara en av dem kan vara aktiv åt gången.

Jag använder de tre olika varianterna i applikationen på dessa ställen:

Type **global** => my-eye-marketing.jsx – som menyknappar.

Type **local** => collapsable-fieldset.jsx – för att “toggla” öppning och stängnig.
content-browser.jsx – för att visa olika sparade bilder.

Type **local-scope** => canvas-history-panel.jxs – för att aktivera ett “item” för ändring, och här får inte två knappar vara aktiva samtidigt.

Blinking Eye komponenten behöver inte heller bara vara ett blinkande öga. Man kan enkelt med props lägga in två egna bilder som knappen ska växla mellan. Skulle till exempel kunna vara en öppen eller stängd mun eller en kryssruta.

Knappen kan användas ”stand-alone” eller inuti en <button> alt. inuti React Router Link.

För att Blinking Eye ska kunna fungera i helhet är den beroende av React Router (react-router-dom), och även Prop Types (prop-types). Den använder även context för att distribuera vilken ”route/path” som är aktiv.

Lite krångligare att konfigurera type global, men jag har gjort en readme.md fil i mappen som Blinking Eye ligger i, med tydliga instruktioner för hur den ska konfigureras. Dessutom är den väl dokumenterad med JSDoc i koden.

Min tanke, för att tydliggöra återanvändbarheten, var att göra den till ett eget npm-paket, men tiden för det har inte funnits till. Kanske provar senare för att lära mig.

Kodstruktur och felhantering

Jag har inte gjort så stora förändringar i mappstrukturen kring api-anropen, mer än att jag gjort två nya api-kommunikations-moduler för DELETE och POST. Api-kommunikations-modulerna är fortfarande placerade i Model-mappen(logiken), separerade från View för att tydligt särskilja logik från UI.
Dock har jag i helhet ändrat om i kodstrukturen för anropen, så att ett context (layout-database-context.jsx) distribuerar innehåll från api:et till olika komponenter, istället för att varje komponent som är i behov av data gör egna separata anrop.
Skälen till att skapa ett context för en lokalt samlad databas är fler:

• Inte behöva göra flera api-anrop i olika komponenter som behöver samma data. Eftersom anrop kan ta tid så skapar färre anrop en bättre användarupplevelse om en snabbare applikation.
• Att distribuera data genom context är enklare och säkrare än att göra flera olika anrop. All data blir då synkad i alla komponenter.
• Samt färre ställen att felhantera och mer DRY kod.

Fel i anropen till api:et kan uppstå om api:et t.ex. inte är tillgängligt eller någon URL har förändrats eller att en parameter inte finns. Dessa fel hanteras i api-modulerna som returnerar en array med data eller error. Vid korrekt anrop returneras [data, null] och vid fel [null, err]. Samma logik används i alla olika metoder för anrop vilket ger en tydlighet i koden.
Felmeddelanden till användaren hanteras i de olika komponenterna som anropar api:et. Eftersom jag har ett context som samlar layout-data hanterar contextet ett ev. error och vidarebefordrar (state err) det till layout-panel.jsx som meddelar användaren i UI om felet. Jag använder paketet react-notifications för att informera användaren om problem eller lyckade handlingar som de utför.

Min Error Boundry

Jag har två Error-boundry-komponenter. De är döpta till component-error-boundry.jsx, app-error-boundry.jsx och ligger i en egen mapp /Components/ErrorHandeling. Komponenterna

renderar props.children om de inte fångar upp errors i React-trädet, vilket de gör med Reacts class-metod getDerivedStateFromError().
Fångas ett error upp sätts hasError state till true, vilket villkors-renderar en specifik errorsida, med olika information beroende på vart felet uppstått. Detta är varför jag har två, för att fånga upp error på olika platser i React-trädet.
ComponentErrorBoundry fångar upp fel som uppstår i vy-komponenter som renderas ut för användaren. Om en av dem inte kan renderas fångas det upp och visar en specifik error-sida; ComponentsErrorView. Via denna sidan har användaren fortfarande möjlighet att använda andra komponenter/sidor i applikationen. Alternativen som användaren har när denna sida visas, är att försöka uppdatera sidan (knapp) och se om den kommer igång igen. Skulle det inte funka finns det en knapp för att komma till startsidan. Från startsidan kan användaren sedan navigera runt på de sidor/komponenter som inte kraschat. Skulle även startsidan ha ett problem, finns en text som hänvisar till att starta om applikationen i webbläsaren.

Sker det ett fel i till exempel en context-komponent så funkar inte appen över huvud taget. Därför fångas dessa errors upp tidigare i trädet och ger användaren ett helt annat meddelande. I detta fallet att applikationen inte längre är tillgänglig, men att de kan försöka igen lite senare. Möjligheten att försöka med en uppdatering av sidan finns också som en knapp, samt finns det kontaktmöjligheter till Eye Marketing teamet via mail, där användaren uppmanas att skicka information ang. problemet och på det sättet kunna lösa det snabbt, och även kunna ge återkoppling. Kontaktuppgifter finns med i båda errormeddelandena.

Att Error-boundry visar meddelanden med information och möjligheter till lösningar ger användaren en upplevelse av att applikationen vet att något gått fel och att någon jobbar på att lösa det. Fastnar applikationen på en tom sida, skapar det en frustration och maktlöshet hos användaren. Användaren vet inte vad som ska göras eller vad som händer vilket också är dålig marknadsföring för applikationen. Vi vill ha glada användare som får hjälp, inte problem vilket resulterar i positiv marknadsföring.