GREENSWAY

Inlämningsuppgift 4: A Minimum Viable Product

DEN 21 MAJ 2021

WIVIANNE GRAPENHOLT
grwi20ql@student.ju.se

Innehåll

Länk

WiviWonderWoman/GreenSway

Min återanvändningsbara komponent

Modal

Jag har fokuserat på att göra min modal-komponent återanvändningsbar. Syftet är att visa pop-upp meddelande ”framför” resten av UI komponenterna, på ett snyggare och tydligare sätt än till exempel en alert. Den går att användas helt ”fristående” genom att ett klick på knappen ändrar komponentens state och gör den osynlig. Det går även att skicka in funktionalitet via propsen onClick och även använda den som en stateless komponent och utnyttja propsen className för att styra synligheten.

Komponenten är beroende av tillhörande css fil för att få önskat beteende, jag har brutit ut stylingen till en egen CSS-fil och placerat den i en egen mapp tillsammans med komponenten, jag anser att det ökar återanvändning genom att enkelt och överskådligt kunna ändra på tex: färger, animationen. Naturligtvis är komponenten även beroende av React. Jag använder PropTypes för att typ-checka propsen vilket jag inte anser är ett beroende eftersom komponenten skulle fungera lika bra utan.

Jag använder den i min applikation till fallback-message och har för avsikt att även använda den när en användare klickar på en dag i kalendern för att visa och boka tvättider. Jag är faktiskt så nöjd med den att jag förmodligen kommer adoptera den in i framtida projekt.

Kodstruktur och felhantering

MVI

Jag har ärligt talat tappat räkningen på hur många omstruktureringar jag gjort i detta projekt. Men har landat i en slags MVI struktur. ”Lite” inspirerad av strukturen i den inspelade handledningen (Redux-demo) 🙃 Följande förbättringar har implementerats:

Model (api)

Alla API anrop och backend logik har fått flytta till en egen api-mapp som är uppdelad efter vilket API som skall anropas. Jag har separata caller-filer för axios-instanserna med bas URL som importeras till de filer som gör anrop i ett try & catch statement. Jag har även lagt till en barrel-fil som jag anser tillföra ytterligare avgränsning eftersom denna inte exporterar caller-filerna.

Intent (state)

Eftersom jag lagt till Redux så har det tillkommit ytterligare en mapp, state. Som har undermappar för action, reducers och store.

View (components)

React komponenterna har fått flytta in i en components-map som har vissa undermappar.

Anrop från en React komponent går via props funktioner som dispatchar thunk funktioner i /state/actions. Som från ett try & catch statement i sin tur dispatchar actions och de asynkrona funktionerna i /api. Här görs själva anropet och beroende på vad svaret till thunk-funktionen blir dispatchas ytterligare actions. Alla actions hanteras av reducerna som via root-reducern uppdaterar Redux-storen som den anropande komponenten har tillgång till och därigenom tar emot svaret.

Möjliga fel

Jag kan erkänna att jag inte täckt upp för alla fel men jag har en api-error-boundry (se mer under rubriken Min Error Boundry) som fångar det som har dispatchats som hasError , dvs de fel som fångats av try-catch blocken i /api eller /sate.actions. Jag har inte implementerat någon hantering av oväntade svar tex om ett tomt objekt eller tom array returneras mer än att det förmodligen triggar min error-boundry när data inte finns tillgänglig för komponenterna. Jag tänker att det bästa sättet att validera svaren hade varit att själv skriva ett REST-API vilket jag inte har prioriterat. För att förbättra användarupplevelsen och förhindra vissa komponenter att krascha när Redux-state isLoading har jag skapat en loader som visas i stället för komponenten som väntar på sin data.

Min Error Boundry

Design

Jag har faktiskt två komponenter för att hantera fel.

- *En basic error-boundry som ligger längst ut utanför App, det är en klassisk Error-Boundry som använder getDerivedStateFromError och componentDidCatch och som inte är kopplad till Redux-store. Här fångas alla tänkbara fel och ett fallback-message visas med en knapp som bara laddar om applikationen.*
- *Samt api-error-boundry som förmodligen inte anses vara en ”äkta” error-boundry eftersom den är stateless och inte använder någon av de ovannämnda metoderna. Den är i stället en ”Higher Order Component” som reagerar på om Redux-state hasError och är placerad runt Content eftersom det är innanför där det flesta anropen görs, inser först nu att hämta ny användare och sätta email ligger utanför…😬 men dessa fel lär error-boundryn reagera på. Ett fallback-meddelandet renderas ”ovanpå” header och footer, i stället för en tom skärm, eftersom inte hela React-trädet påverkas. När användaren klickar på knappen görs en check av sparad användare och ett nytt anrop görs efter att hasError och isLoading är ”nollställda”.*

Jag har tyvärr pga. tidsbrist inte implementerat någon lösning för upprepade fel.

Användarupplevelse

Den största anledningen till att ha dessa två är just användarupplevelsen, jag tycker det är trevligare att se något av applikationen i bakgrunden när det är möjligt samt att användaren kan se loadern, vilket jag anser ger ett intryck av att det är ett mindre fel som enkelt kan åtgärdas.