Labb 3 - Individuell reflektion

När jag byggde bokningsapplikationen i React valde jag att dela upp lösningen i flera mindre komponenter, där varje steg i flödet representeras av en egen komponent. Exempelvis hanteras datum- och tidsval i en komponent, val av bord i en annan och kunduppgifter i ytterligare en. Fördelen med denna struktur är att koden blir mer överskådlig, eftersom varje komponent ansvarar för en tydlig och avgränsad del av gränssnittet. Det gör det enklare både att förstå hur flödet är uppbyggt och att felsöka eller byta ut en enskild del utan att riskera att förstöra helheten. Samtidigt skapar det en viss mängd boilerplate-kod, eftersom data och logik behöver lyftas upp till en gemensam kontext för att kunna delas mellan stegen.

För att hantera state genom hela bokningsflödet använde jag en React Context. På så sätt kan alla steg i processen nå den gemensamma informationen, som exempelvis vald tid, antal gäster, vilket bord som är markerat och kunduppgifterna. Fördelen med detta tillvägagångssätt är att det blir enkelt att flytta sig mellan stegen utan att tappa data, och att komponenterna kan hållas relativt “dumma” och bara läsa eller skriva till context. En nackdel är att om applikationen växer kan context-objektet bli väldigt stort och svårt att överblicka. Då skulle det kunna vara mer lämpligt att dela upp state i flera context eller använda ett mer avancerat state-hanteringsbibliotek som Redux eller Zustand för att bättre kunna skala.

För att säkerställa att datan som skickas till API:et blir korrekt använde jag TypeScript-typer. Genom att definiera typer för Customer, Table och Booking minskas risken att felaktig eller ofullständig data skickas vidare. När formuläret för kunduppgifter fylls i kan jag exempelvis se till att namn och telefonnummer alltid finns innan anropet görs. Detta ökar tillförlitligheten i hela flödet eftersom användaren inte kan gå vidare med ogiltiga uppgifter, och API:et slipper hantera onödiga fel.

När det gäller felhantering och feedback har jag valt en ganska enkel lösning med grundläggande felmeddelanden. Om API:et returnerar ett fel kan användaren få en tydlig indikation på att bokningen inte gick igenom. I praktiken fungerar detta bra så länge API:et svarar relativt snabbt. Om API:et däremot skulle vara långsamt eller otillgängligt hade upplevelsen kunnat bli sämre. Användaren riskerar då att uppleva applikationen som “frusen”. En förbättring hade varit att implementera laddningsindikatorer, retry-logik och mer nyanserad feedback, till exempel att tala om ifall felet beror på nätverket eller på servern.

Om jag skulle vidareutveckla applikationen med fler funktioner, exempelvis möjligheten att avboka, visa en historik över gjorda bokningar eller stöd för flera restauranger, tror jag att kodbasen ändå står på en ganska stabil grund. Komponentstrukturen gör det möjligt att bygga vidare steg för steg. Dock skulle jag förmodligen vilja refaktorera hur jag hanterar state om applikationen växer kraftigt, för att undvika att BookingContext blir för omfattande. Med en mer modulär statehantering och lite mer avancerad felhantering skulle lösningen vara väl rustad för att klara en större skala.