



# Robins porteføljeside v.1

[Home](#)  
[Om meg](#)  
[Ferdigheter](#)  
[Prosjekter](#)  
[Kontakt meg](#)

Prosjektnavn:

Beskrivelse:

Teknologier:

Dato:    
 Ingen fil valgt

Prosjektlenke:

Kategori:

Status:

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

150 × 150

Beskrivelse  
[Lenke til prosjekt](#)

[Facebook](#) [Instagram](#) [LinkedIn](#) [Github](#) [Twitter](#)[NÅX](#)

© 2024 Robins portefølje

Hadde lagt til styling fra før av.

## 1.3 Vurder komponent komposisjonen

Vurder mulige refaktoreringer som kan forbedre din applikasjon. Du skal kommentere hvilke endringer du kan gjøre og fordeler og ulemper med disse. Du trenger IKKE å utføre endringene da du skal lære enda flere alternativer i kommende leksjoner.

Denne oppgaven er første del av flere refleksjoner som skal gjennomføres.

- Fikk hjelp av ChatGPT til dette. Selv så kom jeg ikke på så mye jeg kunne gjøre, men visste det var ting jeg kunne gjort, og brukte da ChatGPT til å gjøre det mer klart for meg.

## 1. Oppdeling av komponenter (Separation of Concerns)

### Mulig endring:

Jeg kan dele opp komponentene ytterligere hvis de begynner å bli for store eller håndterer for mye logikk. For eksempel, hvis Projects.tsx begynner å håndtere mange oppgaver (visning av prosjekter, filtrering, sortering osv.), kan det være fornuftig å dele denne komponenten inn i flere mindre, som f.eks. en egen komponent for kategorifilteret.

### Fordeler:

- Bedre organisasjon og lesbarhet av kode.
- Forenkler testing av individuelle komponenter.
- Gjør komponentene lettere å vedlikeholde og oppdatere.

### Ulemper:

- Kan føre til flere filer, noe som i begynnelsen kan virke overveldende.
- Noen ganger kan for mye oppdeling gjøre komponentene for fragmenterte.

## 2. State Management (Løfting av state eller bruk av Context)

### Mulig endring:

For øyeblikket håndteres state med `useState` i `App.tsx`. Hvis applikasjonen vokser, kan jeg vurdere å bruke React's Context API eller en tredjeparts state management løsning som Redux for å håndtere state mer effektivt.

### Fordeler:

- Context kan redusere "prop-drilling", der props må sendes mange nivåer ned gjennom komponenttreet.
- Det gjør det lettere å dele state mellom flere komponenter på tvers av applikasjonen.

### Ulemper:

- Context API kan være unødvendig komplisert for små applikasjoner.
- Redux eller andre tredjeparts-løsninger krever mer konfigurasjon og forståelse, og kan være overkill for en liten applikasjon.

## 3. Bruke en Formik eller React Hook Form for formhåndtering

### Mulig endring:

I stedet for å håndtere form-logikk (som i `CreateProject.tsx`) manuelt, kan jeg bruke et bibliotek som "Formik" eller "React Hook Form" til å håndtere skjemaer.

### Fordeler:

- Forenkler håndtering av skjemaer, validering og innsendingslogikk.
- Mindre kode i forhold til å håndtere formstate manuelt.
- Bedre støtte for avanserte skjemaer med innebygde verktøy for validering, feilhåndtering, og reset.

### Ulemper:

- Ekstra avhengigheter og læringskurve for biblioteket.
- For små skjemaer kan det føles som "overengineering".

## 4. Bruk av TypeScript typer og grensesnitt (Forbedret typing)

### Mulig endring:

Applikasjonen bruker allerede TypeScript, men jeg kan være enda strengere med typing, for eksempel ved å definere mer presise typer for funksjoner, props og API-responsen.

### Fordeler:

- Hjelper med å unngå feil og gjør koden mer selv-dokumenterende.
- TypeScript kan forbedre utvikleropplevelsen ved å gi bedre autocompletion og feilmeldinger.

### Ulemper:

- Kan bli for detaljert og tungvint å vedlikeholde dersom typestrukturen er for kompleks.
- Liten læringskurve hvis du ikke er vant til komplekse typer i TypeScript.

## 5. Error Handling og Loading States

### Mulig endring:

Jeg kan legge til mer robust feilhåndtering og vis en loader når data hentes. For øyeblikket kastes en error ved feil, men jeg kan vise en brukerdefinert feilmelding i UI, og kanskje en loader mens data hentes.

### Fordeler:

- Bedre brukeropplevelse ved å vise at noe skjer (henting av data).
- Håndterer feil på en mer brukervennlig måte.

### Ulemper:

- Krever flere komponenter og litt ekstra state-håndtering.

- Kan kreve litt mer logikk i komponentene.

## 6. Refaktorering av API-håndtering (Egen API-fil)

### Mulig endring:

Jeg kan lage en egen fil for API-kall (f.eks. api.js eller api.ts) hvor jeg abstraherer alle fetch-forespørsler ut av komponentene.

### Fordeler:

- Fjerner duplisering av fetch-logikk.
- Bedre organisering av API-kall på ett sted, noe som gjør det lettere å endre API-endepunkter senere.

### Ulemper:

- Litt mer kode for å sette opp i starten.
- For små applikasjoner kan det føles unødvendig.

## 7. Memoization (useMemo, useCallback)

### Mulig endring:

Jeg kan bruke useMemo og useCallback for å memoize verdier og funksjoner, slik at de ikke blir rekalkulert eller gjenskapt ved hver re-render.

### Fordeler:

- Forbedrer ytelse ved å redusere unødvendige re-renders.
- Bra for komponenter som er tunge å gjenskape eller kalkulere.

### Ulemper:

- Kan komplisere koden og føre til at memoization ikke brukes riktig.
- For små applikasjoner med lite state-endringer er ytelsesforbedringen minimal.