GRUPPUPPGIFT

Halvtidskoll – React & TypeScript

Kursvecka 6 – React, TypeScript och React Native

Översikt

Om uppgiften

Den här gruppuppgiften är en halvtidskoll som testar era kunskaper från kursvecka 1–6. Ni bygger en fristående mini-applikation från grunden som övar alla centrala koncept ni lärt er hittills. Uppgiften är designad för att ta cirka 2 timmar.

Tidsåtgång:	Cirka 2 timmar
Gruppstorlek:	4–7 studenter (Boiler Room-grupperna)
Arbetssätt:	Parprogrammering och grupparbete. Dela upp er i par/tre och jobba parallellt.
Leverabler:	GitHub-repo med fungerande applikation i TypeScript + skärmdump av appen

Kunskaper som testas

Uppgiften täcker följande koncept från kursvecka 1–6:

Vecka	Tema	Vad som testas
V1	React-grunder	Komponenter, JSX, props, komponenthierarki
V2	State & Effekter	useState, useEffect, formulär, controlled components
V3	Avancerade Hooks	useReducer, useRef, useMemo, useCallback, Custom Hooks
V4	Context API	useContext, Provider-pattern, global state
V5	TypeScript-grunder	Interfaces, typade props, generics
V6	TypeScript i React	Typade events, utility types, discriminated unions

Uppgiftsbeskrivning: TeamPulse – Gruppens Humörbarometer

Ni ska bygga TeamPulse, en mini-applikation där gruppmedlemmar kan logga sitt dagshumör och sin energinivå. Appen visar en gemensam dashboard med gruppens samlade mående. Tänk på det som en förenklad version av en daglig stand-up – fast visuell och anonym.

Applikationen har tre vyer:

- Check-in-vy – Där användaren loggar humör, energi och en valfri kommentar
- Dashboard-vy – Visar gruppens samlade check-ins för dagen med statistik
- Historik-vy – Listar tidigare check-ins (sparade i state/localStorage)

Varför TeamPulse?

Uppgiften är medvetet frikopplad från BoilerRoom-projektet för att ni ska kunna visa era kunskaper utan att luta er mot befintlig kod. Det testar om ni verkligen förstår koncepten och kan tillämpa dem i ett nytt sammanhang.

Steg-för-steg: Uppdelning i delar

Uppgiften är uppdelad i fyra delar. Varje del bygger vidare på föregående. Tidsuppskattningar är ungefärliga.

Del 1: Projektsetup och datamodell med TypeScript (ca 25 min)

Börja med att sätta upp projektet och definiera era TypeScript-typer. Det här är grunden som allt annat vilar på.

1a) Skapa ett nytt React + TypeScript-projekt med Vite:

```
npm create vite@latest teampulse -- --template react-ts
cd teampulse
npm install
npm run dev
```

1b) Definiera datamodellen i en fil types.ts:

Skapa följande TypeScript-typer och interfaces. Tänk på att använda rätt TypeScript-konstruktioner (union types, interfaces, enums eller string literal unions).

Ni behöver minst följande typer:

- Mood – En union type med minst fem humörnivåer (t.ex. "great", "good", "okay", "tired", "stressed")
- EnergyLevel – Ett numeriskt värde 1–5

- CheckIn – Ett interface med: id (string), name (string), mood (Mood), energy (EnergyLevel), comment (valfri string), timestamp (Date)
- DayStats – Ett interface för dashboard-statistik: averageEnergy (number), moodDistribution (Record<Mood, number>), totalCheckIns (number)

Tips

Använd Record<Mood, number> för moodDistribution. Fundera på om EnergyLevel ska vara en type alias (t.ex. type EnergyLevel = 1 | 2 | 3 | 4 | 5) eller bara number med validering.

Del 2: State-hantering med useReducer och Custom Hooks (ca 35 min)

Nu bygger ni kärnan i applikationen: state-hantering med en reducer och custom hooks.

2a) Skapa en reducer för check-ins:

Implementera en checkInReducer som hanterar följande actions:

- ADD_CHECKIN – Lägger till en ny check-in
- REMOVE_CHECKIN – Tar bort en check-in baserat på id
- CLEAR_DAY – Rensar alla check-ins för dagens datum

Definiera action-typer som en discriminated union i TypeScript:

```
// Exempel på discriminated union för actions
type CheckInAction =
  | { type: "ADD_CHECKIN"; payload: Omit<CheckIn, "id" | "timestamp"> }
  | { type: "REMOVE_CHECKIN"; payload: { id: string } }
  | { type: "CLEAR_DAY"; payload: { date: string } };
```

2b) Skapa custom hooks:

Skapa minst två custom hooks:

useCheckIns – Hanterar check-in-logiken:

- Wrappa useReducer med er checkInReducer
- Exponera funktioner: addCheckIn, removeCheckIn, clearDay
- Returnera aktuella check-ins och funktionerna

useDayStats – Beräknar statistik:

- Ta emot en array av CheckIn som parameter
- Använd useMemo för att beräkna DayStats (genomsnittlig energi, humörfördelning)
- Returnera beräknad statistik

Tips

Använd Omit<CheckIn, "id" | "timestamp"> som payload-typ för ADD_CHECKIN – det testar att ni förstår utility types från vecka 6. Generera id och timestamp i reducern.

Del 3: Context och Provider-arkitektur (ca 30 min)

Nu lyfter ni state-hanteringen till global nivå med Context API.

3a) Skapa en CheckInContext och CheckInProvider:

Implementera en Context som exponerar:

- checkIns – Alla check-ins (CheckIn[])
- todayCheckIns – Filtrerade check-ins för idag (använd useMemo)
- stats – Dagens statistik (från useDayStats)
- addCheckIn, removeCheckIn, clearDay – Funktionerna från useCheckIns

3b) Skapa en ThemeContext och ThemeProvider:

Implementera en enkel tema-hantering med:

- theme – "light" | "dark"
- toggleTheme – Funktion som växlar tema
- Använd useCallback för att stabilisera toggleTheme

3c) Skapa custom hooks för context-konsumtion:

Skapa useCheckInContext och useTheme som wrappar useContext med felhantering:

```
// Exempel: custom hook för context-konsumtion
export function useCheckInContext(): CheckInContextType {
  const context = useContext(CheckInContext);
  if (!context) {
    throw new Error(
      "useCheckInContext måste användas inom CheckInProvider"
    );
  }
  return context;
}
```

3d) Wrappa App-komponenten med providers:

```
// I main.tsx eller App.tsx
<ThemeProvider>
  <CheckInProvider>
    <App />
  </CheckInProvider>
</ThemeProvider>
```

Del 4: Komponenter och UI (ca 30 min)

Nu bygger ni de tre vyerna som använder er Context och era hooks.

4a) CheckInForm-komponent:

Bygg ett formulär med controlled components som låter användaren:

- Skriva in sitt namn (text input)
- Välja humör (knappar eller dropdown – typat med Mood)
- Välja energinivå 1–5 (range slider eller knappar)
- Skriva en valfri kommentar (textarea)
- Skicka in formuläret (anropar addCheckIn via Context)

Krav på formuläret:

- Alla inputs ska vara controlled (useState för formulärstate)
- Typade event handlers: React.ChangeEvent<HTMLInputElement>, React.FormEvent<HTMLFormElement> etc.
- Validering: namn och humör måste fyllas i innan submit
- Formuläret ska nollställas efter inskickning

4b) Dashboard-komponent:

Visa dagens samlade statistik från Context:

- Antal check-ins idag
- Genomsnittlig energinivå (avrunda till en decimal)
- Humörfördelning (visa antal per humör, t.ex. med enkel tabell eller lista)
- Lista alla dagens check-ins med namn, humör-emoji, energi och eventuell kommentar

4c) History-komponent:

Visa alla tidigare check-ins sorterade efter tid (senaste först):

- Använd useRef för att hålla en referens till listan (t.ex. för scroll-to-top)
- Knapp för att rensa dagens data (clearDay)

4d) Enkel navigation:

Implementera enkel navigation mellan vyerna. Ni behöver inte använda React Router – det räcker med ett state som håller koll på aktiv vy:

```
// Enkel navigation med state (React Router är inte ett krav)
type View = "checkin" | "dashboard" | "history";
const [activeView, setActiveView] = useState<View>("checkin");
```

Bonusutmaningar (om tid finns)

Om ni blir klara med grunduppgiften innan tiden är slut kan ni välja en eller flera av dessa utmaningar:

LocalStorage-persistens

Använd en custom hook useLocalStorage<T> som synkar state med localStorage. Hooken ska vara generisk och typsäker:

```
function useLocalStorage<T>(key: string, initialValue: T): [T, (value: T) =>
void] {
  // Implementera med useState och useEffect
}
```

Emoji-mapping med Record

Skapa en typsäker mapping från Mood till emoji med Record<Mood, string>:

```
const moodEmoji: Record<Mood, string> = {
  great: "🤩",
  good: "😊",
  okay: "😐",
  tired: "😴",
  stressed: "😰",
};
```

Energi-trend med useRef

Använd useRef för att spara föregående energimedelvärde och visa om det gått upp eller ner sedan förra check-in.

Generisk FilterList-komponent

Bygg en generisk komponent som kan filtrera och visa vilken lista som helst:

```
interface FilterListProps<T> {
  items: T[];
  filterFn: (item: T, query: string) => boolean;
  renderItem: (item: T) => React.ReactNode;
  placeholder?: string;
}
```

Bedömningskriterier

Uppgiften bedöms inte med betyg, men använd följande checklista för att självutvärdera ert arbete:

Kriterium	Koppling	✓ / ✗
Komponenterna är uppdelade i små, återanvändbara delar med tydliga props	**V1**	
Props skickas korrekt genom komponenthierarkin	**V1**	
useState används för formulärstate med controlled components	**V2**	
useEffect används korrekt (t.ex. för localStorage-synk)	**V2**	
useReducer hanterar check-in-state med tydliga actions	**V3**	
Minst en custom hook är skapad och används	**V3**	
useRef används för DOM-referens eller persisterat värde	**V3**	
useMemo används för beräknad statistik	**V3**	
Context API används för att dela data globalt	**V4**	
Provider-pattern med custom hook för context-konsumtion	**V4**	
Alla typer definierade med interfaces/type aliases	**V5**	
Props-interfaces för alla komponenter	**V5–V6**	
Typade event handlers (ChangeEvent, FormEvent)	**V6**	
Discriminated union för reducer actions	**V6**	
Utility types används (Omit, Record, Partial)	**V6**	
Inga TypeScript-varningar (strict mode)	**V5–V6**	

Föreslagen tidsplan

Anpassa efter ert arbetssätt, men här är en rekommenderad fördelning:

Tid	Aktivitet	Detaljer
0–10 min	**Planering**	Läs igenom uppgiften. Diskutera och fördela arbetet. Bestäm vem som gör vad.
10–35 min	**Del 1: Setup & typer**	Skapa projektet, skriv alla TypeScript-typer i types.ts. Alla i gruppen bör förstå typerna.
35–70 min	**Del 2: Reducer & hooks**	Implementera reducer och custom hooks. Testa att logiken fungerar med console.log innan ni går vidare.
70–95 min	**Del 3: Context**	Skapa Context-providers och wrappa appen. Verifiera att data flödar korrekt.
95–120 min	**Del 4: Komponenter**	Bygg UI-komponenterna. Koppla ihop allt. Testa hela flödet.

Tips för grupparbetet

Fördela arbetet parallellt! T.ex. kan ett par jobba med types.ts och reducer (Del 1–2) medan ett annat par börjar med Context-strukturen (Del 3) och ett tredje par skissar på komponenterna (Del 4). Synka regelbundet så alla är på samma spår.

Rekommenderad mappstruktur

Här är en föreslagen mappstruktur för projektet:

```
teampulse/
├── src/
│   ├── types.ts                // Alla TypeScript-typer
│   ├── App.tsx                 // Huvudkomponent med navigation
│   ├── main.tsx                // Entry point med providers
│   ├── hooks/
│   │   ├── useCheckIns.ts      // Custom hook för check-in-logik
│   │   ├── useDayStats.ts      // Custom hook för statistik
│   │   └── useLocalStorage.ts  // (Bonus) Generisk localStorage-hook
│   ├── context/
│   │   ├── CheckInContext.tsx   // Context + Provider för check-ins
│   │   └── ThemeContext.tsx     // Context + Provider för tema
│   ├── components/
│   │   ├── CheckInForm.tsx     // Formulär för att logga humör
│   │   ├── Dashboard.tsx       // Dagens statistik och översikt
│   │   ├── History.tsx         // Historik över check-ins
│   │   ├── Navigation.tsx      // Navigationsbar
│   │   └── MoodBadge.tsx       // Liten komponent för humör-visning
│   └── reducers/
│       └── checkInReducer.ts   // Reducer + action-typer
└── package.json
```

Avslutning

När ni är klara, diskutera följande frågor som grupp:

- Vilka delar kändes självklara och vilka var svåra?
- Var TypeScript till hjälp eller hinder under utvecklingen?
- Hur kändes det att arbeta med useReducer jämfört med flera useState?
- Om ni fick göra om det – vad hade ni gjort annorlunda?
- Hur kan ni ta med era lärdomar tillbaka till ProFlow-projektet?

Sammanfattning

Den här uppgiften samlar ihop sex veckors lärande i ett litet men komplett projekt. Ni har övat på komponenter och props, state med useState och useReducer, custom hooks, Context API med Provider-pattern, och TypeScript genom hela stacken. Bra jobbat – ni är halvvägs genom kursen!