Avancerad JavaScript

Föreläsning 11

NORTH LINK

Laboration 3

- Deadline för labben är på fredag den här veckan

react-transition-group

- I React hanterar vi vad som visas på skärmen genom data som finns i komponenternas state (eller skickas ner som props)
- Om vi vill göra en animation när något tas bort behöver vi på något vis hålla kvar det i state medan animationen körs
- En metod kan vara att använda **setTimeout** och det går även att lägga till en eventlyssnare på elementen som ska animeras och vänta på “transitionend”
- Detta kan resultera i väldigt invecklad kod och det skapar onödigt komplicerad logik i våra komponenter
- Biblioteket **react-transition-group** innehåller komponenter som gör det enklare och mer stabilt att skapa transitions

react-transition-group

- **react-transition-group** var tidigare en del av React men har brutits ut till en separat modul
- Biblioteket är baserat på ng-animate som är en väldigt populär modul till AngularJS
- Komponenterna lägger automatiskt till och tar bort klasser för sina barn, så att vi kan skriva CSS-regler till dem

https://reactcommunity.org/react-transition-group/

CSSTransition

- CSSTransition kan användas för att göra en animation när ett element visas eller döljs
- CSSTransition tar ett antal props men ärver även props från en mer grundläggande komponent som heter Transition

https://reactcommunity.org/react-transition-group/transition

https://reactcommunity.org/react-transition-group/css-transition

CSSTransition - Exempel

- För att använda CSSTransition behöver vi ange ett antal props
  - **classNames** används för att ange vilka klassnamn som ska läggas till
  - **timeout** används för att ange hur många millisekunder animationen tar
  - **unmountOnExit** används om vi vill ta bort elementet från DOM:en efter det döljts
  - **in** är en Boolean som bestämmer om elementet ska visas eller döljas

```
import { CSSTransition } from 'react-transition-group';

<CSSTransition
  classNames="message"
  timeout={200}
  unmountOnExit
  in={this.state.showMessage}
>
  <div className="message">Hello</div>
</CSSTransitions>
```

CSSTransition - Exempel

- CSSTransition kommer automatiskt lägga till klasser på sitt barn, vilka namn klasserna får beror på vad vilket värde vi sätter **classNames** till
- Om vi sätter classNames till “message” kommer följande klassnamn läggas till
  - **message-enter** läggs till när elementet ska visas
  - **message-enter-active** läggs till direkt efter **message-enter**
  - **message-exit** läggs till när elementet ska döljas
  - **message-exit-active** läggs till direkt efter **message-exit**
- I nästa slide finns ett exempel på hur de här klasserna kan användas för att skapa en fade-animation

CSSTransition - Exempel

```
.message-enter {
  opacity: 0;
}
.message-enter-active {
  opacity: 1;
  transition: opacity 0.2s;
}
.message-exit {
  opacity: 1;
}
.message-exit-active {
  opacity: 0;
  transition: opacity 0.2s;
}
```

Övning - Dialogruta

1. Skapa en ny React-applikation
2. Installera react-transition-group
3. Skapa en knapp och använd CSSTransition för att visa en dialogruta med en animation när användaren klickar på knappen
4. Dialogrutan ska ha en knapp som går att använda för att stänga den

Portals

- Ibland vill vi rendera element utanför applikationens rot-element
- T ex brukar modala dialogrutor renderas i ett element som ligger längst ner i body-taggen för att kunna positionera dialogrutan ovanför alla andra element på sidan
- I React brukar man använda en teknik som kallas portals för att kunna rendera React-komponenter utanför applikationens rot-element

https://reactjs.org/docs/portals.html

TransitionGroup

- Något vi gör väldigt ofta i webb-applikationer är att rendera ut listor
- Det går ofta att lägga till och ta bort element i listorna
- I React kan det vara ganska omständigt att implementera animationer när vi tar bort element ur en lista eftersom vi måste hålla kvar datan i state tills animationen är klar
- **react-transition-group** innehåller en komponent som heter TransitionGroup som hjälper oss med detta
- Den används tillsammans med CSSTransition och kommer automatiskt sätta **in** prop till **true** eller **false** när element läggs till och tas bort

TransitionGroup - Exempel

```
import { TransitionGroup, CSSTransition } from ‘react-transition-group’;

<TransitionGroup>
  {items.map(item => (
    <CSSTransition key={item.id} classNames=”item” timeout={200}>
      {/* Rendera ett element här */}
    </CSSTransition>
  ))}
</TransitionGroup>
```

TransitionGroup

- TransitionGroup i föregående exempel gör att vi inte behöver hålla koll på vilka element i listan som ska visas eller döljas, utan den reagerar automatiskt när något element läggs till eller tas bort
- Vi kan använda samma CSS-kod som när vi använder CSSTransition på enstaka element

Övning - Tabell över personer

1. Skapa en React-applikation med en tabell som visar personer
2. Tabellen ska innehålla kolumner för namn, ålder och email-adress
3. Lägg till ett formulär så att det går att lägga till nya personer i tabellen
4. Varje rad i tabellen ska ha en knapp för att ta bort en person
5. Använd TransitionGroup och CSSTransition för att lägga till en animation när personer läggs till och tas bort från listan

Laboration 4

- Laboration 4 finns i PingPong
- Deadline för laborationen är 26 april kl 08:59
- Den här labben skiljer sig ganska mycket från tidigare labbar. Den här gången ska vi skapa ett Fyra i rad-spel i React

Fyra i rad

- Fyra i rad är ett populärt sällskapsspel
- Det heter Connect Four på Engelska
- Spelet går ut på att släppa ner cirkelformade brickor i kolumnerna på spelplanen
- Spelplanen är ett rutnät med 7 kolumner och 6 rader
- En spelare vinner när fyra brickor av samma färga ligger i en linje (vertikalt, horisontellt eller diagonalt)

NORTH LINK

Fyra i rad

- Er uppgift är att implementera spelet med React
- Er applikation ska visa spelplanen och spelarna släpper ner brickor genom att klicka på kolumnerna
- När en spelare släpper en bricka behöver ni kontrollera att det fortfarande finns plats i kolumnen
- Om det finns plats behöver ni räkna ut vilken rad brickan hamnar på
- Efter att en spelare lagt en bricka behöver ni kontrollera om spelaren har vunnit. Om spelaren inte vunnit går turen över till nästa spelare
- Om spelplanen fylls upp utan att någon spelare vunnit blir resultatet lika

Fyra i rad

- Det mest komplicerade momentet kommer att bli att räkna ut om en spelare har vunnit
- Försök att komma fram till en enkel algoritm för att räkna ut om fyra brickor av samma färg ligger i rad
- Det finns fyra olika fall
  - Horisontellt
  - Vertikalt
  - Diagonalt nedåt åt höger
  - Diagonalt uppåt åt höger
- Spelet ska visa ett meddelande när en spelare vunnit eller om resultatet är lika
- När en runda är över ska en knapp för att starta om spelet visas

Krav

- Spelet ska vara skrivet med React
- Spelet ska visa ett 7x6-rutnät av cirklar
- Logiken för när spelare släpper brickor ska vara implementerat korrekt. Det ska inte gå att släppa en bricka i en full kolumn
- Logiken för att kontrollera om en spelare vunnit ska vara implementerad korrekt
- Spelet ska visa ett meddelande när en spelare vunnit eller resultatet är lika
- En knapp för att starta om spelet ska visas när spelet är över

Tips

- Testa gärna att implementera applikationen med React hooks. T ex genom att använda **useReducer** (vi ska börja titta på hooks idag)
- Börja med att implementera rutnätet och logiken för att släppa brickor. Koden för att kontrollera vem som vunnit kan skrivas sist och oberoende av resten av programmet
- Använd penna och papper för att komma fram till en smart regel innan ni försöker implementera logiken för att kontrollera vem som vunnit

React Hooks

- React hooks introducerades i React 16.8
- Med hooks kan vi skriva komponenter med tillstånd utan att skriva klasser
- Läs gärna följande introduktion till hooks som även ger en motivation till varför hooks har lagts till
  https://reactjs.org/docs/hooks-intro.html
- React innehåller ett antal inbyggda hooks men det går även att skapa hooks själv, s.k. “custom hooks”

useState

- I React 16.8 introduceras ett antal “hooks” som alla är funktioner där funktionsnamnet börjar med “use”
- Den första vi ska kolla på är “useState” som används för att ge state till funktionskomponenter
- Tidigare kallades funktionskomponenter för “tillståndslösa funktionskomponenter” men fr o m React 16.8 är de inte längre tillståndslösa

useState

- **useState** används för att skapa en tillståndsvariabel och används genom att anropa funktionen i en funktionskomponenten
- useState returnerar en array som både innehåller värdet på tillståndsvariabeln och en funktion som används för att uppdatera den
- Som argument används tillståndsvariabelns initiala värde

```
import { useState }, React from ‘react’;

function MyStatefulComponent(props) {
  const [ value, updateValue ] = useState(0); // Initierar en variabel till
värdet 0
  ...
}
```

useState - Exempel

- Här är ett enkelt exempel som använder en tillståndsvariabel för att ändra färg på ett element

https://github.com/argelius/ec-advanced-js-samples/blob/master/lesson11/hooks-use-state/src/App.js

Övning - useState

1. Ska en ny React-applikation med en funktionskomponent
2. Komponenten ska ha ett formulär med radioknappar där användaren kan välja sitt favoritdjur (t ex hundar, katter, hästar, fiskar, etc.)
3. Formuläret ska kontrolleras genom en tillståndsvariabel som skapas med **useState**
4. Skriv ut något i stil med “Jag gillar också hästar” under formuläret

Flera tillståndsvariabler

- **useState** ger oss enstaka tillståndsvariabler och en funktion som uppdaterar en variabel
- Detta skiljer sig lite från klasskomponenter där tillståndet är ett objekt och vi kan uppdatera delar av objektet med **this.setState**
- För att ge en komponent flera tillståndsvariabler går det att anropa useState flera gånger

```
const [color, updateColor] = useState('red');
const [height, updateHeight] = useState(100);
```

Övning - Beräkna BMI

1. Skapa en ny React-applikation
2. Skapa en funktionskomponent, **BodyMassIndexCalculator**, som har ett formulär där man kan mata in en längd (i cm) och en vikt (i meter)
3. Använd längden och vikten för att beräkna ett värde för BMI och skriv ut det på skärmen

Viktigt att tänka på

- Anropa alltid setState (och andra hooks) längst upp i en funktion
- Anropa aldrig hooks inuti if-satser eller loopar
- Detta är viktigt eftersom ordningen som hooks anropas är viktigt

```
const [height, updateHeight = useState(180);
const [weight, updateWeight] = useState(80);
```

- I koden ovan finns det inget annat sätt för React att veta vilken variabel som är vilken förutom ordningen som useState anropades

useEffect

- För komponenter som har sidoeffekter (t ex att de hämtar data från något API eller uppdaterar DOM:en) har det tidigare inte varit möjligt att använda funktionskomponenter
- Med en hook som heter **useEffect** kan vi göra samma saker som vi gör i livscykelmetoder som **componentDidMount** och **componentDidUpdate**

https://reactjs.org/docs/hooks-effect.html

useEffect - Exempel

- useEffect tar en funktion som argument och den funktionen kommer att exekveras både när komponenten skapas första gången och varje gång den uppdateras
- Därför motsvarar **useEffect** både componentDidMount och componentDidUpdate för klasskomponenter

```
function MyComponent() {
  const [data, updateData] = useState(null);

  useEffect(() => {
    fetchData().then(updateData);
  }, []);

  ...
}
```

NORTH LINK

useEffect - Undvika loopar

- Precis som med **componentDidUpdate** är det möjligt att orsaka en oändlig loop med **useEffect**
- Anledningen är att funktionen körs efter varje uppdatering och att en effekt brukar orsaka en uppdatering
- För att motverka detta kan vi skicka en Array som argument till useEffect med alla värden som vi vill ska orsaka att effekten körs om de ändras
- Detta kan användas på samma vis som när vi jämför **prevProps** med **this.props** i componentDidUpdate
- I exemplet på förra sidan finns det inga props som ska orsaka att vi hämtar ny data så vi anger en tom Array som argument

useEffect - Exempel

Ett exempel som hämtar en slumpmässig aktivitet från Bored API och visar på skärmen

https://github.com/argelius/ec-advanced-js-samples/blob/master/lesson11/hooks-use-effect/src/App.js

Övning - useEffect

1. Skapa en application med en funktionskomponent
2. Använd **useEffect** för att göra ett anrop för att hämta alla länder från det här API:et
   https://restcountries.eu/
3. API:et svarar med en lista av alla världens länder. Rendera en tabell med två kolumner, en för namn och en för huvudstad

Hantera uppdateringar

- I förra övningen hämtade vi bara data när sidan laddades
- Det går att använda **useEffect** för att reagera på uppdateringar
- Till exempel, om vi har en sökruta kan vi göra anrop när innehållet i sökrutan uppdateras
- För att undvika att hamna i en oändlig loop behöver vi använda det andra argumentet till useEffect

```
const [query, updateQuery] = useState('');
const [data, updateData] = useState(null);

useEffect(() => {
  if (query.length > 0) {
    fetchData(query).then(updateData);
  }
}, [query]);
```

NORTH LINK

Övning - Sökning

1. Utgå från applikationen ni skapade i förra uppgiften
2. Lägg till en textruta och gör det möjligt att söka på länder med namn
   Filtrera inte listan, utan gör ett anrop till API:et
3. Lägg till debounce på textrutan
   Använd gärna på https://github.com/xnimorz/use-debounce