Avancerad JavaScript

Föreläsning 6

NORTH LINK

Vad vi gjorde förra lektionen

- Funktionskomponenter
- Bra syntax för att skriva React
- Tillståndslösa komponenter

Funktionskomponenter

- Det finns två sätt att skapa komponent i React
- Vi kan skapa komponenter som **klasser** eller som **funktioner**
- Hittills har vi bara tittat på klasskomponenter men det är också väldigt vanligt att skapa komponenter som funktioner

Funktionskomponenter

- För att skapa en komponent som en funktion behöver vi bara skapa en funktion som returnerar ett React-element
- Nedan följer en väldigt enkel komponent

```
const Hello = () => (
  <p>Hej! Jag är en komponent.</p>
);
```

Props för funktionskomponenter

- För klasskomponenter är props definierade som **this.props**
- För funktionskomponenter skickas istället props som argument till funktionen

```
const Greeting = (props) => (
  <p>Hello, {props.name}!</p>
);
```

State i funktionskomponenter

- Det går inte att använda **state** i funktionskomponenter på samma vis som vi gör i klasskomponenter
- I React 16.8 introducerades något som heter React Hooks som låter oss använda state i funktionskomponenter men det är något vi ska kolla på senare i kursen

Livscykelmetoder i funktionskomponenter

- Det går inte heller att använda **componentDidMount** och **componentWillUnmount** i funktionskomponenter så vi kan inte kontrollera komponentens livscykel på samma vis som vi gör med klasskomponenter
- Också det här går att göra i funktionskomponenter med React Hooks!

Varför använda funktioner istället för klasser?

- Om vi bortser från React Hooks verkar klasskomponenter vara mycket kraftfullare än funktionskomponenter
- Det är sant men det finns ändå vissa fördelar med att skriva komponenter som funktioner som t ex
    - Mindre kod. Både mindre kod att skriva och mindre kod att skicka till webbläsaren.
    - Funktionskomponenter utan något tillstånd är lättare att förstå för den som läser koden eftersom de bara renderar ut props och andra komponenter
    - Om vi försöker skriva så många funktionskomponenter som möjligt tvingar det oss även att isolera state till några få komponenter vilket gör våra applikationer mer skalbara

Tillståndslösa komponenter

- Eftersom vi vill undvika att dela state över många komponenter kommer stora applikationer i allmänhet att bestå av väldigt många komponenter som inte har något eget state alls
- Det är viktigt att känna sig bekväm med att skriva tillståndslösa komponenter
- Eftersom det inte finns något state måste all data som komponenter visar komma som props från föräldern
- Eftersom vi inte kan ändra state i någon förfader från komponenten måste vi anropa funktioner som kommer som props från föräldern (som i sin tur kan ändra state)

Exempel - Tillståndslös komponent

- I det här exemplet finns en tillståndslös komponent MoodPicker
- Den kan användas för att välja vilket humör man är på men den kontrolleras helt av props från föräldern
- Humöret skickas genom en prop som heter **currentMood**
- Om användaren klickar på någon av radioknapparna kommer **onChange** att anropas med en sträng som argument så att det går att reagera på ändringen i föräldern

https://github.com/argelius/ec-advanced-js-samples/blob/master/lesson05/mood-picker/src/MoodPicker.js

Övning - Carousel

1. Skapa en komponent som heter **Carousel**
2. Komponenten ska visa en bild i taget och användaren ska kunna byta bild genom att klicka på cirklarna under bilden
3. En av cirklarna ska var ifylld så att det går att se vilken bild som visas
4. När bilden ändras ska en ny bild visas genom att bilderna animeras till vänster eller höger
5. Komponenten ska inte ha något eget state utan ska kontrolleras utifrån genom props.
6. (extra) Gör så att man kan byta bild genom att "swipea" vänster och höger på en mobiltelefon
   Använd t ex biblioteket Hammer.js

NORTH LINK

Vad vi ska göra idag

- Ni kommer att få tillgång till laboration 2
- CRUD och REST-API:er
- Routing i React

Laboration 2

- Labb 2 finns på PingPong
- Deadline är fredag 29 mars
- Uppgiften går ut på att använda ett REST-API för att lista, lägga till, redigera och ta bort filmer
- Applikationen ska skrivas som en SPA med React och ni behöver implementera routing

REST-API

- Det finns ett REST-API tillgängligt på

http://ec2-13-53-132-57.eu-north-1.compute.amazonaws.com:3000/movies

- Det stödjer följande HTTP-verb
  - **GET** för att lista och hämta specifika filmer
  - **POST** för att lägga till nya filmer
  - **PUT** för att ändra information om filmer
  - **DELETE** för att ta bort filmer

REST-API

- API:et skickar endast data som JSON
- När ni lägger till nya filmer eller redigerar filmer behöver ni skicka data med JSON
  Det är viktigt att använda korrekt Content-Type!

Objekt-struktur

- Objekten som ni får från API:et och objekten ni skickar till API:et ska ha följande struktur

```
{
  “id”: “99a7d7ba-8660-4011-b98c-c927c5f0d34c”,
  “title”: “A title”,
  “description”: “A description”,
  “director”: “A director”,
  “rating”: 3.0,
}
```

Validering

Servern gör följande validering på objekten

- “title” ska vara en sträng mellan 1 och 40 tecken lång
- “description” ska vara en sträng mellan 1 och 300 tecken lång
- “director” ska vara en sträng mellan 1 och 40 tecken lång
- “rating” ska vara ett nummer mellan 0.0 och 5.0
- “id” skapas av servern och kan inte modifieras.

API-referens

- **GET /movies**
  Svarar med en lista av filmer och statuskod 200

- **GET /movies/:id**
  Svarar med en film som har ett specifikt **id** och statuskod 200

  Om filmen inte finns svarar den med statuskod 404

- **POST /movies**
  Lägger till en ny film. Datan ska skickas som JSON.

  Svarar med statuskod 400 om objektet är ogiltigt. Om filmen läggs till svarar den med statuskod 201

API-referens

- **PUT /movies/:id**
  Uppdaterar en film som har ett specifikt **id**. Datan ska skickas som JSON.

  Om filmen inte finns varar den med statuskod 404. Om objektet är ogiltigt svarar den med statuskod 400.

  Om den lyckas uppdatera filmen svarar den med statuskod 200.

- **DELETE /movies/:id**
  Tar bort en film. Svarar med statuskod 404 om filmen inte existerar.

  Om den lyckas ta bort filmen svarar den med statuskod 204.

Vyer

Applikationen ska innehålla fyra vyer

- En huvudsida som visar en tabell med filmer
- En sida där användaren kan lägga till nya filmer
- En sida där användaren kan redigera en film
- En sida där användaren kan se information om en viss film

Huvudsida

- Huvudsidan ska visa en tabell med filmerna som hämtas från API:et
- Det kommer aldrig finnas mer än 20 filmer sparade så pagination behövs ej
- Filmtitel, regissör och betyg ska visas i tre olika kolumner.
  Rendera gärna betyget på något roligt sätt. T ex som stjärnor
- Varje rad i tabellen ska ha tre knappar/länkar
  - En knapp för att ta bort filmen
  - En länk till redigeringssidan
  - En länk till informationssidan
- Huvudsidan ska även innehålla ett textfält som används för att filtrera filmerna på **titel** och **regissör**
  API:et stödjer ej filtrering så filtreringen måste ske på klienten

Sida för att lägga till filmer

- Det ska finnas en sida där användaren kan lägga till nya filmer.
- Sidan ska innehålla ett formulär med följande element
  - Ett textfält för **titel**
  - En textarea för **beskrivning**
  - Ett textfält för **regissör**
  - Någon typ av element för betyget. Det kan t ex vara en “range”-input eller något roligt element som t ex att man kan klicka på antal stjärnor
- När användaren skickar formuläret ska något av följande hända
  - Om servern svarar med någon typ av fel ska ett felmeddelande visas
  - Om filmen läggs till ska användaren skickas till huvudsidan

Redigeringssida

- Den här sidan används för att uppdatera en film
- Den ska fungera som sidan där man lägger till filmer men formuläret ska automatiskt fyllas i

Informationssida

- Den här sidan ska visa titel, beskrivning, regissör och betyg för en film
- Den ska även innehålla en länk till redigeringssidan

Routing

- Applikationen ska implementera korrekt routing
- Den ska innehålla minst fyra “routes”, en för varje vy
- Webbläsarens historik ska fungera korrekt (man ska kunna gå bakåt och framåt)
- När man laddar om sidan ska man stanna kvar på samma sida
- Sidans titel ska ändras dynamiskt när användaren går till en ny sida

Krav

- Applikationen ska vara en SPA skriven med React
- Den ska innehålla fyra vyer
  - Huvudsida
  - Sida för att lägga till en film
  - Sida för att redigera en film
  - Informationssida för en film
- Den ska implementera korrekt routing
- Den ska hantera fel från API:et korrekt

Tips

- Använd biblioteket **react-router** för att implementera routing i applikationen. Det är ett väldigt populärt bibliotek och har en bra dokumentation
- Använd biblioteket **react-helmet** för att uppdatera titeln på sidorna
- Försök att dela upp applikationen i små, enkla delar och implementera dem var för sig
- Implementera varje sida som en liten applikation med sitt eget state
- Hjälp varandra och fråga om hjälp om ni fastnar med något!
- Lycka till!

CRUD

- När man pratar om datalagring brukar man prata om fyra grundläggande operationer
  - Create - Lägga till ny data
  - Read - Hämta data
  - Update - Uppdatera data
  - Delete - Ta bort data
- De här operationerna brukar förkortas **CRUD**
- När vi skriver en applikation som ska kunna hantera och manipulera data behöver vi implementera alla dessa funktioner

CRUD och HTTP

- I HTTP kommunicerar vi med en webb-server med hjälp av verb som GET och POST
- GET används för att hämta en resurs (Read)
- POST används för att skapa en ny resurs (Create)
- PUT används för att ersätta en resurs (Update)
- PATCH används för att ändra en resurs (Update)
- DELETE används för att ta bort en resurs (Delete)

REST

- **REST** står för Representational state transfer
- Det är det absolut vanligaste sättet att designa API:er för webben
- I REST används HTTP-verben tillsammans med URL:er för att utföra olika operationer

REST - GET

- I REST används verbet GET för att hämta en lista av objekt eller ett specifikt objekt
- Om vi har en resurs som vi kallar “movies” kan vi använda GET på följande vis

```
GET /movies
```

- Servern kommer att svara med en lista över filmer
- Om vi tänker att alla filmer har ett unikt id kan vi även hämta en specifik film med

```
GET /movies/star-wars
```

REST - POST

- POST används för att lägga till en ny resurs i ett REST-API
- Själva datan brukar skickas som JSON men i vissa fall som XML
- Så för att lägga till en ny film används

```
POST /movies
```

- Datan som skickas till API:et brukar kallas för “payload”
- När datan skickas brukar API:et svara med statuskoden 201 (Created) för att berätta för klienten att allt gick bra
- Om servern inte accepterar det som skickats svarar den med statuskod 400 (Bad request)

REST - PUT

- PUT används för att uppdatera en viss resurs. Datan som skickas är samma som för post
- Så om vi vill uppdatera en viss film kan vi göra

```
PUT /movies/star-wars
```

- Precis som med POST kommer servern att svara med statuskoden 400 (Bad request) om det är något problem med datan som skickats
- Om allt går bra svarar servern med statuskod 200

REST - DELETE

- För att ta bort resurser i ett REST-API används DELETE

```
DELETE /movies/star-wars
```

- Om servern lyckas ta bort resursen svarar den oftast med statuskod 200 (Accepted), eller i vissa fall 204 (No Content)

Routing i React

- Till skillnad från ramverk som Angular har React inget inbyggt stöd för routing
- Därför måste vi antingen ta hjälp av ett tredjepartsbibliotek eller skriva logiken själva om vi vill ha routing i våra applikationer
- Det mest populära biblioteket för routing i react är **react-router**
- Det finns även en router som heter **universal-router** som är ganska populär, speciellt för projekt som använder Server Side Rendering
- Vi kommer att använda **react-router** i den här kursen

React Router

- React Router är helt deklarativ, vilket innebär att det inte finns något imperativt API, utan den används helt genom att lägga in komponenter i sin sida.
- Routern går att använda med både React Native för att skriva mobilappar och på webben. När vi använder React Router på webben behöver vi installera en modul som heter **react-router-dom**

```
npm install react-router-dom
```

Router-komponenten

- För att använda React Router behöver vi lägga till en “Router”-komponent i vår applikation
- I **react-router-dom** finns två komponenter vi kan använda: BrowserRouter och HashRouter
- När vi använder create-react-app kan vi använda BrowserRouter men om vi bara har en webb-server som servar filer måste vi använda HashRouter
- Router-komponenten läggs ovanför vår applikation

```
import { BrowserRouter as Router } from 'react-router-dom';

const App = () => (
  <Router>
    {/* Vår applikation här! */}
  </Router>
);
```

Route-komponenten

- För att lägga in Routes i vår applikation används Route-komponenten
- Den tar ett antal props
  - exact - Används om man vill att sökvägen ska matcha exakt
  - path - Används för att ange sökvägen
  - component - Används för att ange vilken komponent som ska renderas för routen

```
import { Route } from 'react-router-dom';

...

<Route exact path="/" component={Home} />
<Route path="/page1" component={Page1} />
<Route path="/page2" component={Page2 />
```

Länkar

- Länkar skapas med Link-komponenten
- Precis som de andra komponenterna kan den importeras från react-router-dom

```
<Link to=”/”>Home</Link>
```

Övning - React router

1. Skapa en ny React-applikation
2. Installera **react-router-dom**
3. Skapa en sida som har en navigation med tre länkar, t ex
    a. Home
    b. Page1
    c. Page2
4. Implementera routes för de här tre länkarna
5. Kontrollera att routingen fungerar som den ska. Kan du ladda om sidan? Kan du gå framåt och bakåt?

React Helmet

- När vi har många sidor i vår applikation behöver vi även kunna ändra titeln på sidan.
- Vi vill ändra innehållet i <title>-elementet så att webbläsarens tabbar ändras
- Vi hade kunnat göra det i **componentDidMount** och det fungerar utan problem
- Ofta vill man ändra mer än titeln när användaren navigerar till en ny sida och det finns ett bra verktyg som heter React Helmet som hjälper oss att göra det
- Andra saker som vi kanske vill kunna ändra är favicon och meta-taggar

https://github.com/nfl/react-helmet

React Helmet

- React Helmet är precis som mycket annat i React en komponent som vi kan rendera någonstans i vår applikation.
- Den kommer att skriva över det som finns i <head>-elementet och det kvittar var i applikationen vi renderar den
- Om vi renderar mer än en Helmet-komponent kommer den senare att gälla (längre ner i trädet eller ett senare syskon)

React Helmet - Exempel

```
import { Helmet } from ‘react-helmet’;

const Home = () => (
  <>
    <Helmet>
      <title>Home</title>
    </Helmet>
    <h1>Home</h1>
    <p>This is the home page</p>
  </>
);
```

Övning - React Helmet

1. Utgå från applikationen ni skapade i föregående övning
2. Installera React Helmet
3. Använd React Helmet för att ändra titeln på era sidor
4. Fungerar det som det ska?

Parametrar i routes

- Det går även att definiera routes som accepterar parametrar
- Vanligtvis har man inte bara statiska routes utan vi kan ha routes som visar information om något som hämtas från ett API
- Vi kan lägga in variabler i routes genom att skriva kolon (:) innan dem

```
<Route path=”/users/:id” component={UserDetails} />
```

- Vi kan komma åt parametrarna inuti våra komponenter genom **props.match.params**

```
componentDidMount() {
  getUser(this.props.match.params.id)
    .then(user => this.setState({ user }));
}
```

Övning - Routing med parametrar

1. Vi ska implementera samma övning som vi gjort tidigare men den här gången ska vi använda React och React Router
2. Skapa en applikation med två routes: en sida som visar en lista av öl och en sida som visar information om en viss öl
3. Använd https://api.punkapi.com/v2/beers för att skapa listan
4. Varje element i listan ska vara en länk till en sida som visar information om ölet
5. Använd https://api.punkapi.com/v2/beers/:id för att hämta information om en specifik ölsort
6. Använd React Helmet för att uppdatera titlarna. På detaljssidorna för ölen ska titeln vara namnet på ölen.

Redirect

- Som nämnts tidigare är React Router helt deklarativ och all funktionalitet finns som komponenter
- De flesta router-bibliotek har någon typ av imperativ funktion för att göra en redirect, alltså att skicka användaren till en annan sida
- I React Router görs en redirect genom att rendera en **Redirect**-komponent
- Eftersom vi inte kan göra en redirect imperativt brukar man rendera en Redirect-komponent när något villkor är uppfyllt

```
import { Redirect } from 'react-router-dom';
…

render() {
  if (this.state.finished) {
    return <Redirect to="/" />;
  }
  ...
}
```

Exempel - Redirect

- Dokumentationen för React Router har ett väldigt bra exempel för hur en Redirect-komponent kan användas för inloggning

https://reacttraining.com/react-router/web/example/auth-workflow

Switch-komponenten

- Om det finns flera Routes som matchar en sökvägen kommer vanligtvis alla renderas
- I vissa fall vill man att bara en sida ska renderas och då kan man använda Switch-komponenten

```
import { Switch, Route } from 'react-router-dom';

<Switch>
  <Route path="/about" component={About} />
  <Route path="/:user" component={User} />
</Switch>
```