



React (nieuw)

React is een front end library om gebruikersinterfaces mee te bouwen. Het is in 2011 ontwikkeld door Facebook om de code van hun grote webapplicaties beter beheersbaar te maken. Al vrij snel maakten zij de bibliotheek open source: toegankelijk voor het publiek. React is sindsdien nog steeds het populairste front end framework onder developers.

In deze cursus pas je alle kennis die je hebt opgedaan tijdens de cursus JavaScript en HTML & CSS toe om zo een volwaardige webapplicatie te bouwen. Je leert hoe je webpagina's opdeelt in modulaire componenten en hoe je styling toevoegt in zo'n grote applicatie. Bovendien zal je leren hoe de Life Cycles van React werken en hoe je deze gebruikt om jouw webapplicatie te voorzien van interactie en externe data.

Lernoutkomsten

De student bouwt een interactieve en modulaire webapplicatie in React en maakt hierbij gebruik van herbruikbare interface elementen, state management en de life cycles van React.

Prestatie-indicatoren

- De student schrijft schone React code door clean code principles toe te passen.
- De student deelt zijn applicatie op in herbruikbare componenten waarbij data wordt doorgegeven via properties.
- De student voorziet componenten van interactie en externe data door gebruik te maken van de React Life Cycle hooks.
- De student gebruikt modulaire styling die aan specifieke componenten gekoppeld is.

1. React: wat is het?

1.1 Inleiding

React is een front end library om gebruikersinterfaces mee te bouwen. Het is in 2011 ontwikkeld door Facebook om de code van hun grote webapplicaties beter beheersbaar te maken. Al vrij snel maakten zij de bibliotheek open source: toegankelijk voor het publiek. React is sindsdien nog steeds het populairste front end framework onder developers.

Tot voor kort was het gebruikelijk om alle logica en automatisering te laten plaatsvinden in de back-end. Moest er op jouw website een berekening gemaakt worden? Dan werd er eerst een nieuwe request naar de server gestart. Niet erg handig als de gebruiker snel resultaat verwacht.

JavaScript is in de afgelopen jaren een veel grotere rol gaan spelen in websites en webapplicaties. JavaScript kan veel processen en berekeningen uitvoeren in de browser, zonder met de server te hoeven communiceren. Hierdoor is een applicatie sneller en prettier in gebruik. Maar door de toenemende complexiteit van applicaties is het ook steeds moeilijker geworden om JavaScript te onderhouden.

Dit is waar React om de hoek komt kijken: React gebruikt gestandaardiseerde JavaScript structuren om orde in de chaos te scheppen en code makkelijker te onderhouden. Andere voordelen van deze library zijn:

- Het ontsluiten van een applicatie is ander en efficiënter omdat je herbruikbare User Interface componenten kan maken, zonder bestaande code te herschrijven;
- Een applicatie is schaatsbaar en makkelijk uit te breiden met plugins;
- Code is netter en beter te onderhouden;
- Er is een enorme offline compatibility waarbij de mogelijkheden kunnen vinden op je vragen;
- Alle componenten zijn reactief en kunnen direct geactualiseerd worden;
- Het is ten opzichte van andere frameworks zoals Angular relatief makkelijk te leren.

1.2 Component library

React maakt gebruik van User Interface elementen die je kunt hergebruiken. Maar wat bedoelen we daar eigenlijk mee? React componenten lijken op HTML-elementen, zoals bijvoorbeeld een button, waarbij we specifieke functionaliteit toevoegen om ze herbruikbaar te maken. Neem bijvoorbeeld eens een kijkje op deze website:

LOODS 5
SHOP ▾
EIGEN COLLECTIE
SALE
MERKEN & DESIGNERS ▾
RUIMTES ▾
TRENDS ▾
EIGEN HUIS & TUIN
Zoek

Banken

Stoelen

Vloerkleden

Verlichting

Eettafels

Fauteuils

Kasten

Accessoires

Summer Sale!
Tot 50% korting
op ons eigen merk

stoel -30%

kleed 50%

Slaap Deals!
Tot 50% korting
op ESSENZA beddengoed

HKliving stoel Retro Webbing

HKliving | Stoel
€ 249,-

[+](#) [S](#)

HKliving kamerscherm Webbing

HKliving | Decoratie object
€ 449,-

[+](#) [S](#)

Eetkamerstoel Bridges boucle

Loods 5 Goods | Stoel
€ 195,-

[+](#) [S](#)

FÉST Vaas Obi

FÉST | Vaas
€ 29,-

[+](#) [S](#)

FÉST schaal Dixon black

FÉST | Schaal
€ 39,-

[+](#) [S](#)

We zien dat dezelfde categorie-button vaker terugkomt. Het enige verschil is de tekst op de button, het icoon, en de pagina waar hij naar toe linkt. Vroeger zouden we alle buttons uitschrijven, met ieder een eigen unieke tekst. Met React kunnen we één generiek button component bouwen die telkens andere data accepteert, maar er hetzelfde uitzet en zich op dezelfde manier gedraagt. Super efficient dus!

Op deze manier kun je websites gaan zien als een verzameling van herbruikbare componenten.

LOODS 5

- SHOP
- EIGEN COLLECTIE
- SALE
- MERKEN & DESIGNERS
- RUIMTES
- TRENDS
- EIGEN HUIS & TUIN

Zoek

Banken

Stoelen

Vloerkleden

Verlichting

Eettafels

Fauteuils

Kasten

Accessoires

Summer Sale!
 Tot 50% korting
 op ons eigen merk

stoel
-30%

kleed
50%

Slaap Deals!
 Tot 50% korting
 op ESSENZA beddengoed

HKliving stoel Retro Webbing
 HKliving | Stoel
 € 249,-

+

HKliving kamerscherm Webbing
 HKliving | Decoratie object
 € 449,-

+

Eetkamerstoel Bridges boucle
 Loods 5 Goods | Stoel
 € 195,-

+

FEST Vaas Obi
 FEST | Vaas
 € 29,-

+

FEST schaal Dixon black
 FEST | Schaal
 € 39,-

+

Generische componenten: productblokken (geel), stels met subtitels (roze), categorie-knopen (groen), menu-items (oranje)

Je kunt je voorstellen hoeveel tijd je als ontwikkelaar bespaart als je slechts één categorie-knop, één product blok en één menu-item hoeft te bouwen die je vervolgens kunt hergebruiken op alle pagina's. Beslist de designer openens dat de buttons niet groen, maar roze moeten zijn? Geen probleem! We passen dit aan op één plek en het heeft effect over de gehele webapplicatie.

1.3 JavaScript XML

React maakt gebruik van **JSX**. JavaScript XML. Dit lukt op het eerste gezicht misschien wat abstract, maar eigenlijk is het onze oude vertrouwde HTML in een JavaScript-jasje.

Om te begrijpen hoe JSX en React samenwerken, beginnen we bij het startpunt van elke webpagina: het `index.html` bestand. Als je naar de index van een React project gaat kijken, is het opmerkelijk hoe weinig er eigenlijk in staat:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8"/>
</head>
<body>
<noscript> You need to enable JavaScript to run this app.</noscript>
<div id="root"></div>
</body>
</html>
```

Er staat slechts één `<div>` in de body en die is leeg! Dat betekent niet dat deze applicatie geen pagina's of content bevat, maar hier wordt de volledige React webapplicatie in de browser genereert. Dit heet client-side rendering. Wanneer we dit bestand van de server ontvangen, is de pagina nog leeg. Alle React componenten worden door middel van JavaScript geïnjecteerd. Heeft de gebruiker JavaScript niet aan staan? Dan zullen ze slechts een witte pagina zien.

We moeten onze HTML-elementen door middel van JavaScript op de pagina zien te krijgen, dit doen we met **JSX** (HTML elementen in een JavaScript-jasje). Met JSX kunnen we HTML elementen maken in een volledig JavaScript georiënteerde bibliotheek.

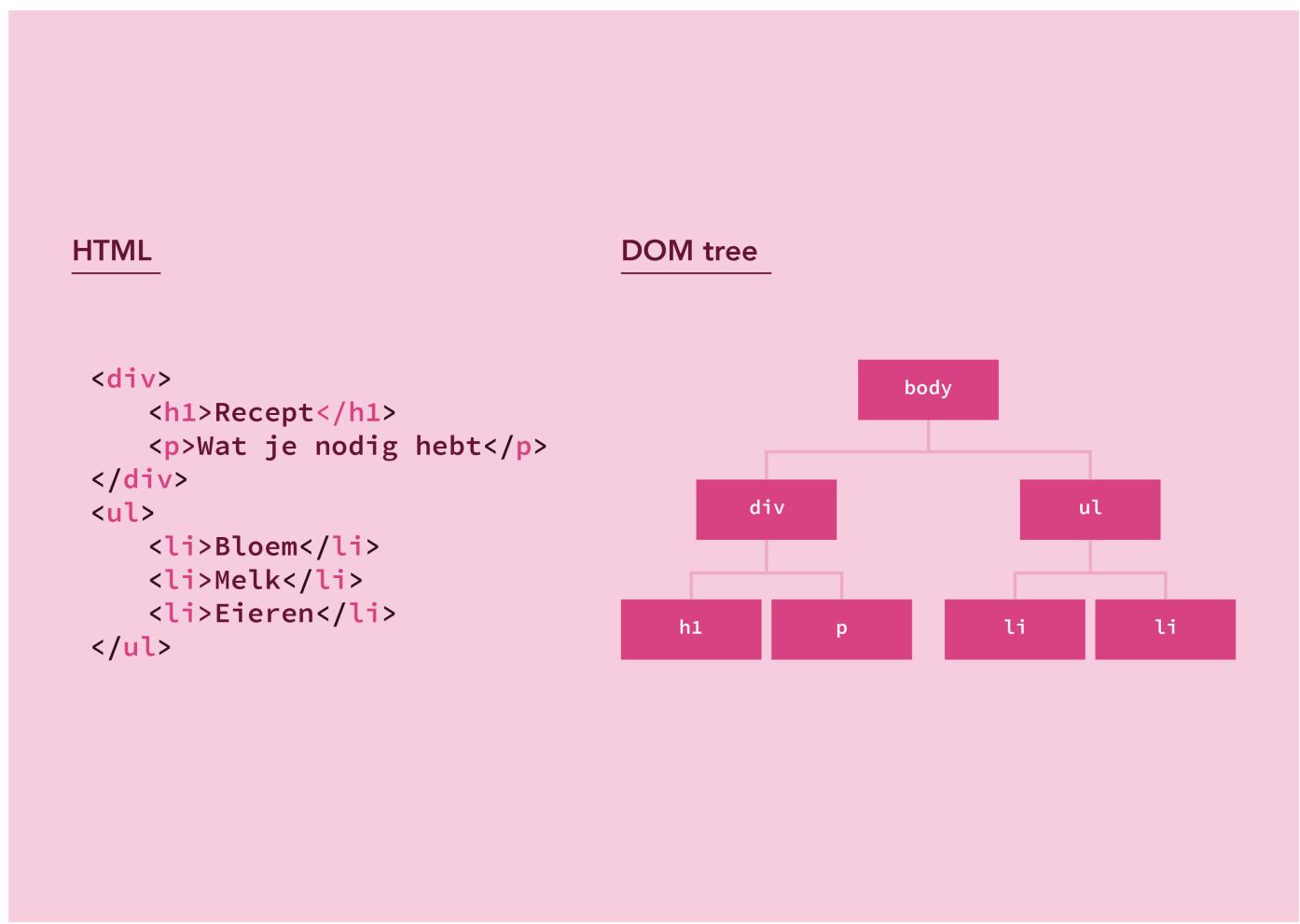
Laten we bij het begin beginnen: we willen een titel op onze webpagina. In het volgende voorbeeld wordt - vereenvoudigd - getoond hoe we een normale `<h1>` tag met React kunnen injecteren in onze applicatie:

```
// We stoppen ons HTML-element in een JavaScript variabele
const pageTitle = <h1>SX is cool</h1>;
// We creëren een referentie naar een element met de id root
const root = ReactDOM.createRoot(document.getElementById('root'));
// We gebruiken de render-methode van React. Deze accepteert één argument, namelijk het element dat wij willen injecteren:
root.render(pageTitle);
```

Elementen worden via deze methode geïnjecteerd in de pagina en ook staandar in een element met de id root. Komt dat je al bekend voor? Precies! Dat is de ene lege `<div>` in onze `index.html`! In dit voorbeeld hebben we alleen een titel gemaakt, in de praktijk injecteren we op deze manier de volledige applicatie het HTML-bestand in.

1.4 De virtuele DOM

Mogelijk heb al kennis gemaakt met Document Object Model van een webpagina in de cursus Javascript. Maar hoe werkt React onder de motorkap? React maakt gebruik van een **virtuele DOM** om een efficiëntieslag te maken ten opzichte van standaard JavaScript.
Iedere keer als een element aangepast moet worden op een webpagina, gebeurt dit via de DOM. Een node wordt toegevoegd, aangepast of verwijderd. Echter, iedere keer als dit gebeurt, moet de browser opnieuw alle CSS berekenen, de layout bepalen en dit op het scherm 'tekenen'. Dat neemt tijd in beslag. Om een webpagina zo snel mogelijk te maken en houden, hebben de makers van React geprobeerd om de hoeveelheid DOM-veranderingen tot een minimum te beperken.



Voor de "echte" DOM creëert React een corresponderende "virtuele DOM", een lichtgewicht kopie van het origineel. De virtuele DOM is licht omdat het eigenlijk een Javascript object is. Dit object kunnen we supersnel uitvoeren of aanpassen. De virtuele DOM van onze pagina ziet er (ongeveer) zo uit:

```
{
  tag: 'body',
  children: [
    {
      tag: 'div',
      children: [
        {
          tag: 'h1',
          children: 'Recept',
        },
        {
          tag: 'p',
          children: 'Wat je nodig hebt',
        }
      ],
    },
    {
      tag: 'ul',
      children: [
        {
          tag: 'li',
          children: 'Bloem',
        },
        {
          tag: 'li',
          children: 'Melk',
        },
        {
          tag: 'li',
          children: 'Eieren',
        }
      ],
    }
};
```

Stel dat we een `class`-attribuut willen toevoegen aan één van onze list items. Onder de motorkap gebeurt het volgende:

- React maakt een koptje van de huidige virtuele DOM en past daarin de elementen aan die zijn veranderd. In ons geval zal deze koptje er bijna hetzelfde uitzien, behalve dat er een `class`-attribuut met de waarde "Ingredient" is toegevoegd.
- Volgens vergelijkt React de originele virtuele DOM met de nieuwe virtuele DOM. Onthoud: omdat dit slechts Javascript objecten zijn, kan dit razend snel.
- Er ontstaat een "diff" object met alleen de verschillen tussen de originele en de nieuwe virtuele DOM. In ons geval staat er alleen een list-item in, met een nieuwe class.
- Als alle veranderingen ("diff's") verzameld zijn wordt de echte DOM aangepast. React weet nu dat alleen de list item opnieuw gegenereerd hoeft te worden en laat de andere lijst-elementen met rust.

Bij het bouwen van jouw React applicatie, ben je zelf gelukkig weinig bezig met de virtuele DOM. Desalniettemin is het wel belangrijk om te begrijpen hoe een React applicatie verschilt van een gewone webpagina.

1.5 Een eigen project opzetten

Het opzetten van een React project ging vroeger handmatig. Dit kon behoorlijk tijdrovend zijn. Gelukkig heeft React hier iets voor bedacht: met een paar simpele commando's wordt er een nieuw project voor je aangemaakt en volledig geconfigureerd. Zo kun jij veel sneller aan de slag!

Om gebruik te maken van deze automatische React installatie heb je een versie van Node.js nodig die hoger is dan 14. Als je Node.js hebt geïnstalleerd tijdens deze leerlijn zit je hier al ver boven.

Kies in WebStorm voor "Nieuw project", en selecteer vervolgens een "React Project". Wanneer je jouw project een naam hebt gegeven, zal de installatie beginnen. Gebruik je een andere editor? Voer het volgende commando handmatig in, in de Command Prompt of Terminal:

```
npx create-react-app <naam-van-jouw-app>
```

Het installeren van het project kan een paar minuten in beslag nemen. Als je de melding happy hacking! krijgt te zien, is het goed gegaan.

```
npm install vs npm start
```

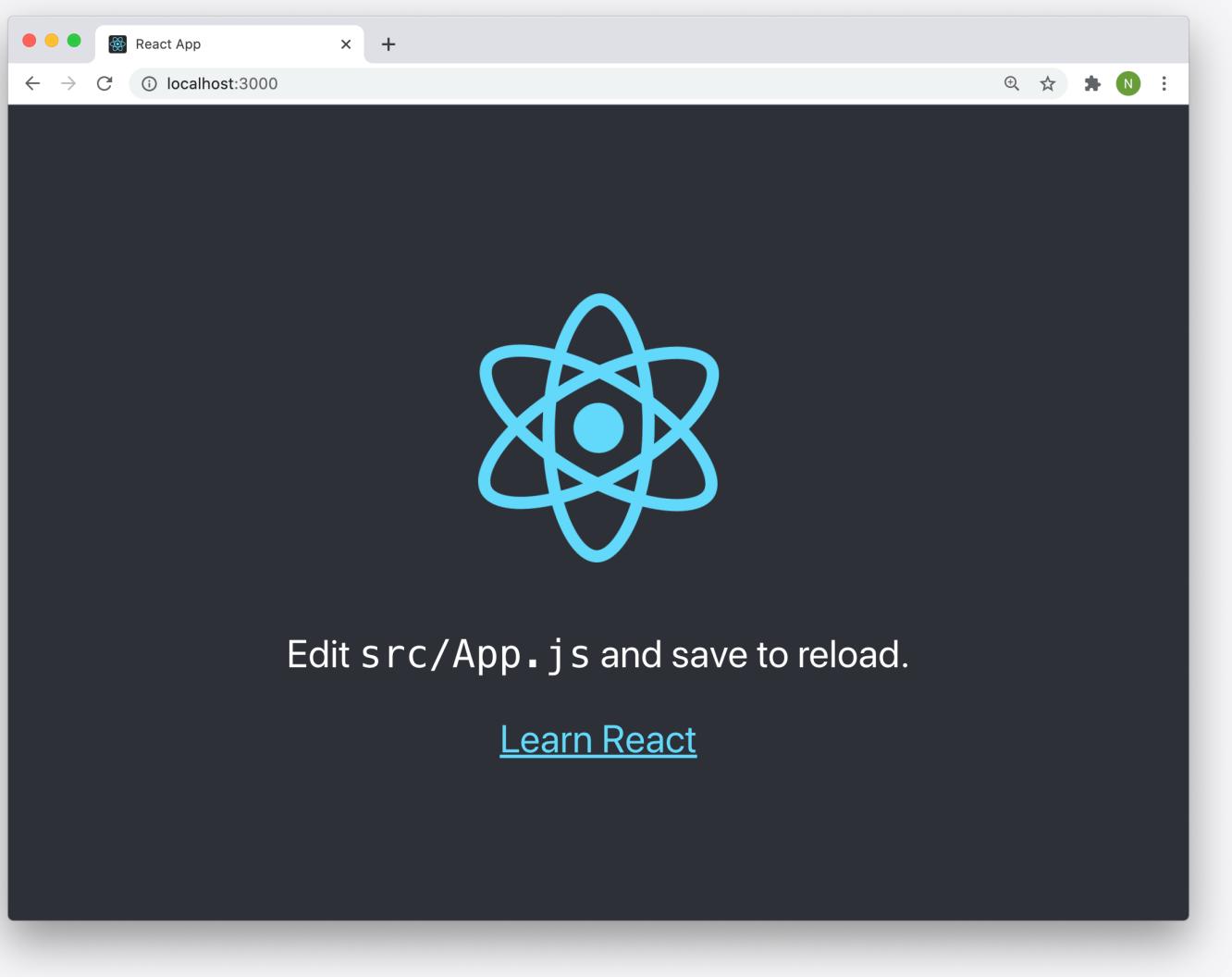
De reden dat het opzetten van het project even lang duurt, komt omdat er heel veel node_modules geïnstalleerd moeten worden. Een React project bestaat namelijk uit allemaal bestanden. Zoals je weet, worden node_modules niet meegepushed naar GitHub en is het niet de bedoeling om die mee te sturen als iemand anders toegang heeft tot jouw project. In de package.json (die je overigens niet zelf hoeft aan te maken!) wordt bijgehouden welke node_modules het project gebruikt, ook wel dependencies genoemd. Wanneer je een project van iemand anders opent, voor je daarom altijd eerst een globale installatie uit:

```
npm install
```

Hiermee haal je alle dependencies (`node_modules`) van het project binnen. Als je je zou voorstellen dat jouw project een auto is, zet je er met `npm install` wielen onder. Dit hoeft je dan ook niet heel vaak te doen: alleen als je de auto voor het eerst gaat gebruiken, of als het niet zo soepel blijkt te rijden. Wanneer je er al vaker in hebt gereden, hoeft je de auto slechts te starten om ermee te gaan rijden. Dit geldt ook voor jouw project, iedere keer als je de webapplicatie wilt gebruiken, hoeft je enkel de applicatie te starten. We willen dat React onze JavaScript XML in het HTML bestand injecteert, zodat de browser dit kan interpreteren en een webpagina laat zien. Hiervoor gebruiken we het universele `npm` commando in de terminal van jouw IDE (of via de UI van Webstorm):

```
npm start
```

De applicatie zal nu automatisch openen in jouw browser op het adres <http://localhost:3000> en de standaard React boilerplate applicatie laten zien:



Wanneer je iets aanpast en opslaat worden de wijzigingen direct toegepast. Je hoeft de pagina niet handmatig te vernieuwen! Echter, wanneer je het project voor het eerst opent, zal je niet direct weten waar je alles kunt vinden en waar je kunt beginnen met programmeren. In onderstaande video leggen we dit uit.

Zie je de video niet verschijnen? Klik dan [hier](#) om de video in een nieuw tabblad te bekijken.

2. Basisprincipes

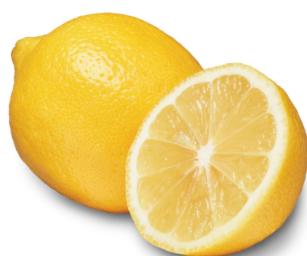
2.1 Inleiding

Je kunt het beste oefenen met React door zelf aan de slag te gaan. We gaan daarom stapsgewijs een simpele webpagina met content maken.

Shop Ons verhaal Blog

Fruit perfection

Shop nu



Citroen

Een citroen is voor de meeste mensen te zuur om zo uit de hand te eten. Van citroen kun je het vruchtvlees, het sap en de schil gebruiken. Het sappige, lichtgele zure vruchtvlees versterkt de smaak van ander voedsel.



Limoen

Limoen is familie van de citroen en de sinaasappel en behoort tot de citrusvruchten (Wijnruitfamilie). Limoenen zijn rond en kleiner dan citroenen. De schil is dun, vrij glad en groen.



Ijsblokjes

Een ijsblokje of ijsklontje is bevroren water in de vorm van een klein blokje. Het wordt gemaakt in een diepvriezer door water in een plastic vorm te laten bevriezen.

Clone de GitHub repository naar jouw computer. Verget niet eerst een npm install te draaien zodat alle benodigde node_modules worden gedownload. Daarna kan je de applicatie opstarten met het commando npm start.

Vergeet tijdens het volgen van de tutorial niet om jouw project op te slaan. Je kunt de wijzigingen pas in de browser zien nadat je ze hebt opgeslagen.

Tip: Aan het einde van elke paragraaf staat een link naar de GitHub branch. Op die branch kun je controleren hoe jouw code er op dat moment uit moet zien.

2.2 HTML en CSS in React

Immiddels weten we dat onze elementen in een HTML-document injecteren via het <App>-component. Hier kunnen we dan ook beginnen met het maken van onze webpagina! Open in jouw editor de src-map en klik het bestand App.js aan. Hierin zie je het volgende staan:

```
import React from 'react';
import './App.css';

function App() {
  return (
    <main>
      Begin hier met de tutorial!
    </main>
  );
}

export default App;
```

De tekst "Begin hier met de tutorial!" wordt door React op de pagina weergegeven: in systeemtaal het dit gerenderd. Plaats binnen de <main>-tag de volgende elementen:

- Een artikel (<article>)
- In dat artikel, de titel "Citroen" (h2)
- ... en een beschrijving: "Een citroen is voor de meeste mensen te zuur om zo uit de hand te eten. Van citroen kun je het vruchtvlees, het sap en de schil gebruiken. Het sappige, lichtgele zure vruchtvlees versterkt de smaak van ander voedsel." (p)
- Het is nu nog een beetje saai... Aanbevelingen lossen dat vast op. Plaats een afbeelding () van een citroen _boven_ de titel. Je kunt deze link gebruiken: https://www.plusonline.nl/sites/plusonline/files/citroen_1.jpg of een image-link van ieder willekeurig plaatje op Google. Hoe de citroen er precies uitziet, is niet zo belangrijk.

Je ziet nu de tekst en afbeelding in de browser verschijnen. Gebeurt dat niet? Dan ben je misschien vergoten de applicatie te starten. Hoe je dit doet, wordt beschreven in de vorige paragraaf. Als het goed is ziet jouw code er nu zo uit:

```
function App() {
  return (
    <main>
      <article>
        <img alt="Citroen" />
        <h2>Citroen</h2>
        <p>Een citroen is voor de meeste mensen te zuur om zo uit de hand te eten. Van citroen kun je het vruchtvlees, het sap en de schil gebruiken. Het sappige, lichtgele zure vruchtvlees versterkt de smaak van ander voedsel.</p>
      </article>
    </main>
);
```

Echter, op jouw pagina vullen die citroenen nu waarschijnlijk je hele beeldscherm... Laten we wat styling toevoegen met CSS!

Hoeveel HTML geen enkel probleem heeft met het woord `class`, mogen we dit in React niet gebruiken. Binnen het JavaScript-landschap is `class` namelijk een reserved keyword, een woord met een speciale betekenis in de syntax van een programmeertaal. In plaats daarvan gebruiken we `className` en niet `class`.

In React gebruiken we `className` in plaats van `class` om styling te voegen

```
HTML: <article className="product">
React: <article className="product">
```

Let op: staat er op één wel `class`, zal er niet zoveel gebeuren. Wanneer je dit op meerdere plekken verkeerd doet, gaat je applicatie rare fouten uithalen!

- Goed: `<article>-tag met de className "product"`
- Goed: de titel (`<h2>Citroen</h2>`) en de beschrijving een `<product-description> className`, zodat de juiste styling wordt toegepast (dit is trouwens geen magie, deze stylingregels hadden we al voor je klaarstaan in App.css)

Kijk eens aan, geen levensgrote citroenen meer!

- Laten we nu andere producten toevoegen in twee extra `<article>`-tags. De tekst hebben we hieronder voor je neergezet, voor de afbeelding mag je de citroen hergebruiken.

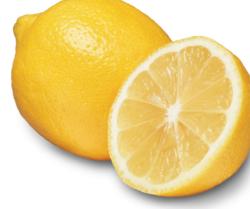
Limoen

Limoen is familie van de citroen en de sinaasappel en behoort tot de citrusvruchten (Wijnruitfamilie). Limoenen zijn rond en kleiner dan citroenen. De schil is dun, vrij glad en groen.

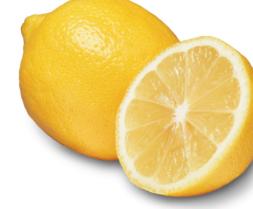
Ijsblokjes

Een ijsblokje of ijsklontje is bevoren water in de vorm van een klein blokje. Het wordt gemaakt in een diepvriezer door water in een plastic vorm te laten bevriezen.

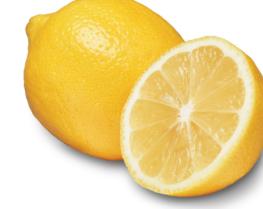
Als het goed is ziet jouw webpagina er nu zo uit:



Citroen



Limoen



Ijsblokjes

Een citroen is voor de meeste mensen te zuur om zo uit de hand te eten. Van citroen kun je het vruchtvlees, het sap en de schil gebruiken. Het sappige, lichtgele zure vruchtvlees versterkt de smaak van ander voedsel.

Limoen is familie van de citroen en de sinaasappel en behoort tot de citrusvruchten (Wijnruitfamilie). Limoenen zijn rond en kleiner dan citroenen. De schil is dun, vrij glad en groen.

Een ijsblokje of ijsklontje is bevoren water in de vorm van een klein blokje. Het wordt gemaakt in een diepvriezer door water in een plastic vorm te laten bevriezen.

Fragments

Oké, het is tijd voor een header bovenaan onze pagina. We gaan een <header>-tag boven onze <main>-tag plaatsen! We moeten zowel de <header> als de <main>-tag samen in één omhullende tag zetten, anders crasht onze applicatie. Waarom? Simpel: App is een functie. Je kunt volgens de regels van JavaScript niet twee verschillende dingen tegelijk teruggeven uit een functie, binnen een return statement.

We moeten de elementen omwikkelen (ofwel wrappen), dit kunnen we doen met een <div>. Echter, gezien we geen extra styling gaan plaatsen, zorgt dit voor een onnodig, extra element. Hier heeft React iets op bedacht: **fragments**. Fragments zijn tags die je gebruikt voor het wrappen van elementen. Ze worden gebruikt om het den-return-value-probleem op te lossen, want ze worden niet op de pagina gerenderd of in de DOM gezet. Zo blijft onze JSX netjes en opeengedrukt.

Simpel, niet?

- Voeg een <header>-tag boven de <main>-tag en omwikkel deze elementen samen in één fragment. Als het goed is zie je nu direct een header-afbeelding verschijnen. Dat is goed tevreden: dat komt omdat we van tevoren door middel van CSS een achtergrondafbeelding hebben toegevoegd aan het header element.
- Plaats in het header-element een met "React perfect" (<h1>)
- Plaats onder de titel een van type button met de tekst: "Shop nu"

Je hebt zojuist een modus beginnen aan jouw eerste React applicatie. Ziet de pagina er niet uit zoals je verwacht, of wil je simpelweg even checken of alles klopt? Je kunt [hier](#) (branch: step-1) bekijken hoe App.js er nu uit zou moeten zien.

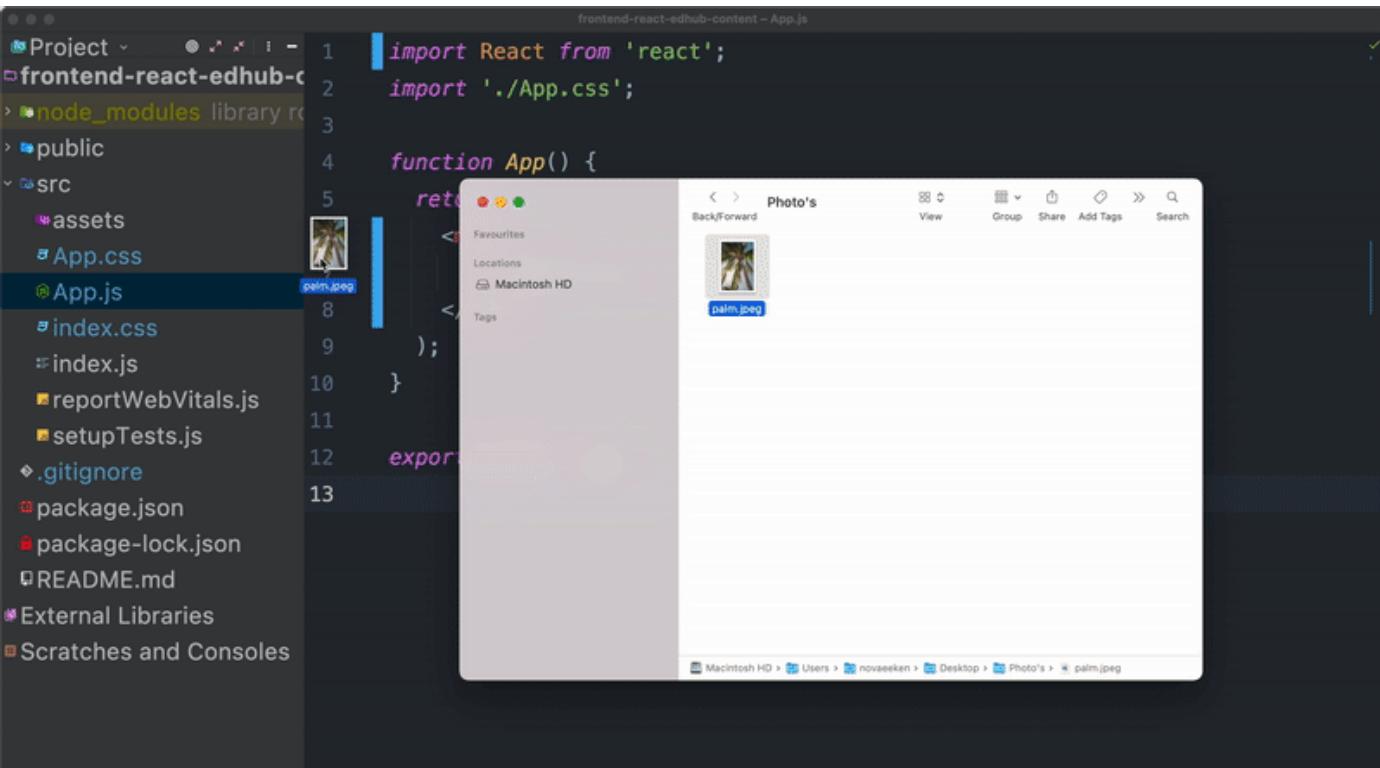
2.3 Eigen afbeeldingen en iconen gebruiken

Je hebt tijdens de cursus HTML & CSS al ondervonden dat je afbeeldingen van het internet makkelijk via een image-link kunt weergeven op jouw webpagina. Maar in de meeste gevallen wil je jouw eigen afbeeldingen gebruiken in een project. Hoe je dit tot nu toe hebt gedaan in HTML, verschilt een klein beetje van hoe je dit doet in React.

Laat we hier direct mee oefenen. In de assets-map van het project vind je drie afbeeldingen voor onze producten:

• citroenen.jpeg
• limoenen.png
• palm.jpg

Het is convertible om de map met beeldmateriaal (video's, afbeeldingen, etc.) assets te noemen, maar uiteraard mag je elke naam kiezen die jij past vindt. Mocht je in de toekomst meer afbeeldingen aan dit project willen toevoegen, kun je deze simpelweg vanaf jouw computer naar de assets-map in WebStorm slepen:



De logische stap zou nu zijn om de afbeelding op de volgende manier te gebruiken, toch?

```

```

Fout! In een React project moet **iedere bron eerst bovenaan in het bestand geïmporteerd worden**, anders kan het niet worden uitgelezen. Geen zorgen, het is heel simpel. We bedenken een naam voor onze afbeelding (niet als bij een variabele) en importeren deze uit de assets map:

```
import citroenen from './assets/citroenen.jpeg';
```

Let erop dat je geen hoofdletters gebruikt in de import-naam. Deze citroenen-variable kunnen we nu gebruiken om de lange link die we meegeven aan onze -tags te vervangen voor de referentie naar onze afbeelding:

```
import citroenen from './assets/citroenen.jpeg';
function App() {
  return (
    <img src={citroenen} alt="Citroenen" />
  );
}
```

Dat ging makkelijk! Maak jij dit al voor de limoen- en palmboom-afbeeldingen? Tip: kijk bij het importeren altijd goed naar de bestandsextensie van de afbeelding waar je mee werkt.

Je kunt de uitgewerkte versie [hier bekijken](#) (branch: stap-2).

SVG's

Een Scalable Vector Graphic (SVG) is een uniek afbeeldings-formaat. In tegenstelling tot normale afbeeldingen die opgebouwd zijn uit pixels, wordt een SVG opgebouwd op basis van 'vector' data. Dit zorgt ervoor dat je oneindig ver in kunt zoomen, zonder dat de afbeelding ooit kortsigt wordt.

Raster **Vector**



Een SVG wordt vaak gebruikt voor bedrijfslogo's en iconsets, het is daarom belangrijk om te weten hoe je ermee werkt. Zo willen wij bijvoorbeeld een winkelwagen-icon in de navigatiebalk. Laten we eerst de juiste elementen klaarzetten:

- Plaats een -tag bovenin de header met daarin:
- Een unieke id (=id), met daarin:
- drie items :
- ... die elk een anchor tag <a>-bevat. De link mag nu voor gewoon naar onze home-pagina wijzen (""), Zet hierin "Shop", "Ons verhaal" en "Blog".

Het winkelwagen-icon willen we achter de unorderde lijst in de navigatiebalk plaatsen. Maar hoe doen we dit? In de *assets*-map hebben we dit icon al (<winkelmandje.svg>) voor je klaar staan. Jij denkt nu: "easy!" Die importeer ik gewoon zo:

```
import shoppingCart from './assets/winkelmandje.svg';
```

FOUT! Nouja, half fout: in theorie zouden we een SVG als afbeelding kunnen gebruiken, maar dan verliezen we alle schaaltbare eigenschappen van het SVG bestand. Dat willen we niet. We moeten SVG's daarom importeren als component:

```
import { ReactComponent as ShoppingCart } from './assets/winkelmandje.svg';
```

Het is belangrijk om hoofdletters te gebruiken bij het `ReactComponent` en bij de gekozen naam. Bij het gebruik van kleine letters herkent React het niet als een component. Om het icon na de unorderde lijst in de navigatiebalk te plaatsen gebruiken we de onderstaande syntax. Let op: het verschilt nog niet direct in de browser.

`<ShoppingCart/>`

Een SVG wordt opgebouwd op basis van 'vector' data. De breedte en hoogte moet gedeclareerd worden, anders kan de SVG niet worden opgebouwd. Dit handelen we af in CSS, door het element een `width` en `height` te geven. Wij hebben deze styling regels al voor je klaar gezet in `App.css`. Jij hoeft alleen nog de `className="shopping-cart-icon"` op het SVG-component te plaatsen. Onthoud dat wanneer je met SVG-bestanden werkt, je altijd de hoogte- en breedte properties toewist in CSS!

Je kunt de uitwerkingen [hier](#) (branch: stap-3) bekijken.

2.4 Event listeners

We hebben inmiddels een prachtige button in onze header staan, maar deze button doet nu nog niets. Laten we beginnen met het weergeven van een bericht in de console, wanneer er een klik-event heeft plaatsgevonden.

• Declareer een functie genaamd `logClick()`, die het bericht "You clicked" in de console logt. Je zet 'm binnen het <App>-component, maar wel buiten de `return`-statement. Zoals hier:

```
function App() {
  function logClick() {
    console.log('You clicked!');
  }
  return (
    <div>
      <h1>React</h1>
      <ul>
        <li>Home</li>
        <li>Shop</li>
        <li>About</li>
      </ul>
      <button onClick={logClick}>Klik mij</button>
    </div>
  );
}
```

```
    /* Elementen staan hier */
}
```

In een JavaScript project zouden we nu eerst een referentie naar het button-element moeten opslaan, om er vervolgens een event en een functie aan te koppelen:

```
const buttonReference = document.getElementById('button');
buttonReference.addEventListener('click', logClick);

function logClick() {
  console.log('You clicked!');
}
```

Voor een React applicatie is dit een stuk makkelijker! We hoeven geen referenties meer op te slaan, want we plaatsen het event namelijk direct op het element zelf via het `onClick`-attribuut:

```
<button type="button" onClick={logClick}>
  Shop nu
</button>
```

Op deze manier kunnen we `onBlur`-events of formulieren plaatsen of `onChange`-events op invoervelden. Bij het aanroepen van events geldt hetzelfde principe als in JavaScript: als de compiler ronde haken achter een functienaam ziet staan, zal de functie direct worden uitgevoerd zonder te wachten op het event. Geef de event listener daarom altijd alleen een functienaam mee - of, als je een aparte functie hebt - plaats de code dan in een anonieme functie:

```
<button
  type="button"
  onClick={() => console.log("Jij wil shoppen!")}>
  Shop nu
</button>
```

Plaats deze event-listener op de 'Shop Nu'-knop in jouw project. Open vervolgens de console op onze webpagina (`rechtermuisknop > inspecteren`) en klik op de 'Shop Nu'-knop.

Echter keert als op de button klikt, wordt de tekst opnieuw gelogd! Wil je de code nog even bekijken? Dat kan je [hier](#) (branch: step-4) doen.

2.5 Verdieping: imports en exports

Code splitting is in ieder soort project een belangrijk onderwerp, vooral in grotere projecten met React. Een gemiddeld stukje software bevat simpelweg te veel regels code om alles in één bestand te zetten: developers kunnen het op die manier niet goed overzien of onderhouden. De code bestaat uit talloze afhankelijkheden, waardoor we uiteindelijk alles met elkaar moeten verbinden. Dit doen we door code uit het ene bestand te exporteren en vervolgens in het andere bestand weer te importeren. Hierin onderscheiden `default exports` en `named exports` het belangrijk om de verschillen goed te herkennen.

Default import- en export

Een default export betekent dat we hetgeen dat we exporteren als 'standaard' beschouwen. We kunnen een default export maar eenmalig in het bestand gebruiken, het is namelijk een hoofd-export. Stel, we hebben een functie geschreven in het bestand `calculateFunctions.js` en willen deze exporteren:

```
function sum(a, b) {
  return a + b;
}

export default sum;
```

Hiermee zeggen we: als je iets importeert uit dit bestand, krijg je standaard de functie `sum`. De import-statement in het verzamel-bestand waar we deze functie willen gebruiken, ziet er dan zo uit:

```
import sum from './helpers/calculateFunctions';
```

We hebben onze functie hier opnieuw `sum` genoemd, maar dat hoeft niet. Omdat we deze functie als standaard exporteren, is de naam niet belangrijk. Het maakt niet uit hoe we de import noemen, want deze zal altijd de default export bevatton. Je mag de import een andere naam geven als dat juist beter uitkomt:

```
import banaan from './helpers/calculateFunctions';
```

Named import- en export

Soms exporteren we niet één, maar meerdere dingen uit één bestand. Je kunt slechts één onderdeel exporteren als standaard, maar hebt daarna oneindig veel `named exports` voor alle andere onderdelen. Zie het volgende voorbeeld:

```
export const pi = 3.14159265359;
export const radius = 76;

function sum(a, b) {
  return a + b;
}

export default sum;
```

Zoals je ziet herken je een named export aan het woordje `export`, in plaats van `export default`. Uit dit bestand exporteren we nu de functie `sum` als default, maar ook twee `named variabelen` (`pi` en `radius`). Wanneer we één of meerdere van deze variabelen willen importeren, moeten we in dit geval wel de originele namen aanhouden. Dit is de enige manier om aan te duiden welke waarden we willen hebben:

```
import { radius } from './helpers/calculateFunctions';
// of:
import { radius, pi } from './helpers/calculateFunctions';
```

Je herkent een named import aan de accolades. Wil je toch de variabelen hernoemen? Dat kan! We doen dit op dezelfde manier zoals je de destructuring gewend bent:

```
import { radius as banaan } from './Helpers/calculateFunctions';
```

Hiermee zeggen we: importeer de named export `radius`, maar gebruik `radius` onder de naam `banaan`. Passt herken je dit nog? Dit heb je al eerder gedaan bij het importeren van de SVG-afbeelding!

2.6 Componenten en properties

Bekijk `App.js` nog eens goed. Als je naar de code kijkt zie je dat het `<App>` component eigenlijk gewoon een functie is, die `JSX` teruggeeft. De functies die we schrijven om React componenten te maken, lijken veel op de JavaScript functies die we al kennen. Ze verschillen van elkaar op twee cruciale punten:

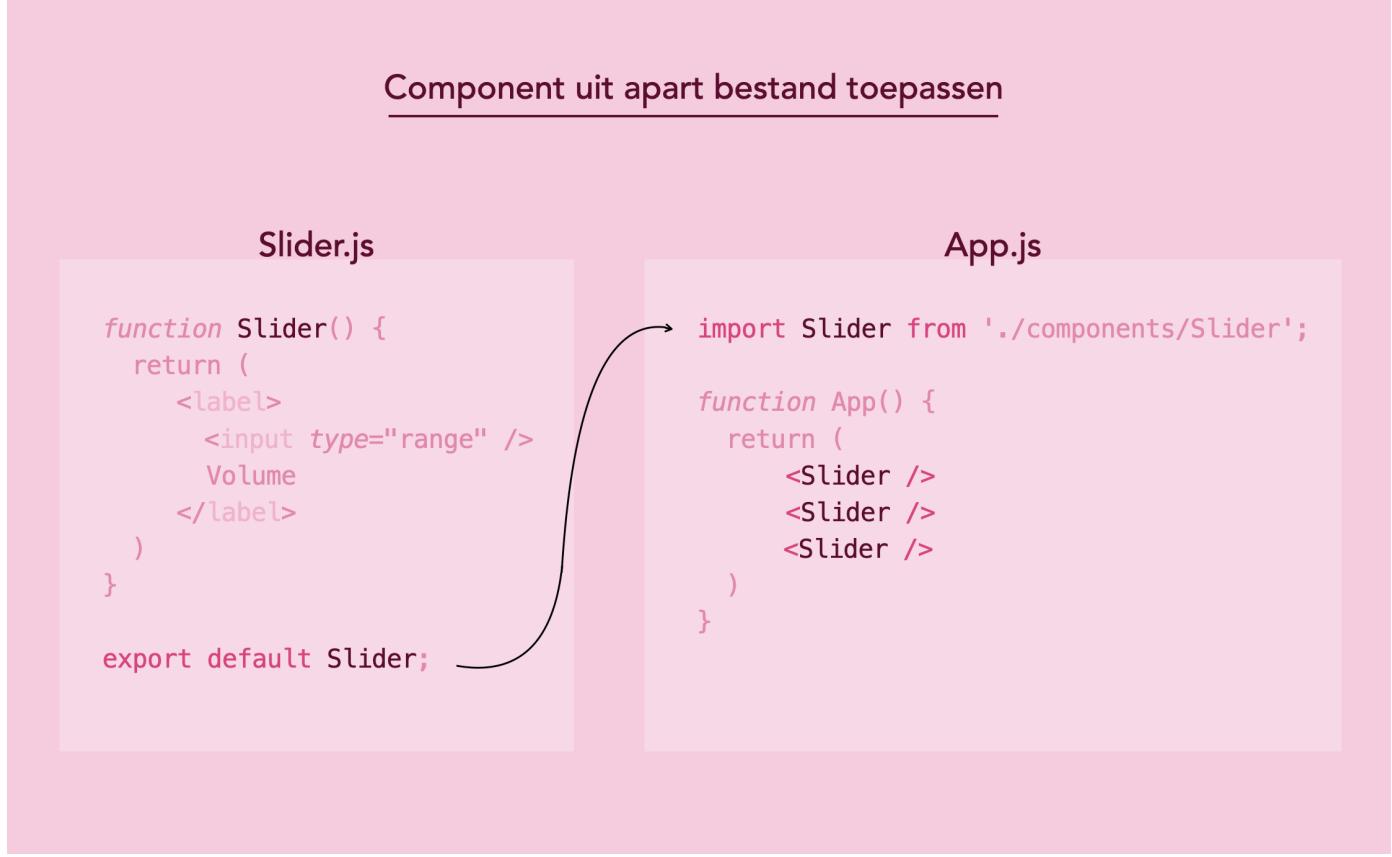
- De functie voor een React component wordt altijd met een hoofdletter geschreven:
- De functie voor een React component *geeft HTML-elementen terug*. Een normale JavaScript functie zal altijd primitieve of structurele datatypes teruggeven, zoals booleans, strings, of een object. De `return`-statement van een React component bevat altijd HTML-elementen: deze worden door React op de webpagina gezet.

Gedurende deze cursus maak je talloze React componenten. We hebben ze nodig voor kleine, herbruikbare onderdelen (zoals de productblokken van Loods5 uit [Hst. 1.2](#)), maar ook voor alle toekomstige pagina's binnen de webapplicatie:

```
function HomePage() {
  return (
    <div>
      <h1>Loods5</h1>
      <p>Welkom op deze website</p>
    </div>
  );
}

function AboutPage() {
  return (
    <div>
      <h2>Over deze winkel</h2>
      <p>Loods5 is opgezet door...</p>
    </div>
  );
}
```

Wanneer we een nieuwe pagina of herbruikbaar component bouwen, plaatsen we dit altijd in een apart bestand. Zo kunnen we de code uit dat bestand (het component) zo vaak als we willen - opnieuw gebruiken. Om het component te gebruiken, importeren we het in `App.js` en schrijven het alsof het een self-closing HTML-element is, maar dan met een hoofdletter:



Properties

Een herbruikbare slider is natuurlijk super handig, alleen is deze slider niet zo herbruikbaar als dat het lijkt. In plaats van drie identieke volume-sliders, willen wij namelijk een volume-, bass- en treble-slider. In de huidige situatie zouden we daar nu twee extra componenten voor moeten bouwen... Dat levert ons alleen maar extra werk op. De HTML voor deze sliders is namelijk nogal eens identiek, met een paar hele kleine verschillen:

- De volume slider heeft een `volumeLabel` en een range van 0 tot 11;
- De bass slider heeft een `bassLabel` en een range van 0 tot 4;
- De treble slider heeft een `trebleLabel` en een range van 1 tot 5.

Gelukkig kunnen we deze waarden doorgeven aan ons slider component door middel van **properties**: zelfbedachte eigenschappen van het component. Dit lijkt op de manier waarop we attributen aan HTML-elementen doorgeven, alleen mogen we de property-namen helemaal zelf verzinnen. Je kunt de property-namen zien als placeholders voor data die wordt doorgegeven. Een logische naam voor onze range-waarden en label zou bijvoorbeeld `textLabel`, `minRange` en `maxRange` zijn. Kijk maar naar het volgende voorbeeld:

```
function App() {
  return (
    <Slider textLabel="volume" minRange="0" maxRange="11"/>
    <Slider textLabel="bass" minRange="0" maxRange="4"/>
    <Slider textLabel="treble" minRange="1" maxRange="5"/>
  );
}
```

Alle properties die wij meegeven worden verzameld in een object. Het Slider-component ontvangt alle properties, dit doen we door de parameter `props` aan de functie-declaratie toe te voegen:

```
function Slider(props) {
```

Wanneer we de properties willen weergeven in ons component, doen we dat door de keys op het props-object aan te spreken. Omdat het nu een variabele betreft en geen letterlijke waarde, gebruiken we accolades:

```
function Slider(props) {
  return (
    <label>
      <input type="range" min={props.minRange} max={props.maxRange}>
        {props.textLabel}
      </label>
  );
}

export default Slider;
```

In React gebruiken we `**` voor strings, en `{}` voor data

De namen die je hebt bedacht in App.js, zoals `minRange`, moeten overeenkomen met de namen die je gebruikte in het Slider-component. Heb je geen zin om het woordje `props` steeds opnieuw te typen? Dan mag je de properties ook direct destructuren:

```
function Slider({ minRange, maxRange, textLabel }) {
  return (
    <label>
      <input type="range" min={minRange} max={maxRange}>
        {textLabel}
      </label>
  );
}

export default Slider;
```

Onderstaande video demonstreert het maken en gebruiken van een herbruikbaar React component. Het maken van componenten met properties is één van de belangrijkste aspecten van React. Zorg ervoor dat je dit concept onder de knie krijgt!

Zie je de video niet verschijnen? Klik dan [hier](#) om de video in een nieuw tabblad te bekijken.

2.7 Drie producten, één component

Laten we het Fruit Perfection project er weer bij pakken. Naast het feit dat al onze producten eruit zien als citroenen, hebben we nu ook een hoop gekopieerde code. De opbouw van al onze artikel tags is namelijk hetzelfde - het enige verschil is de content. Dat kan slimmer. Hier kunnen we iets aan doen door een apart `<Product>`-component te bouwen en die te hergebruiken:

- Maak een nieuwe bestand en noem deze `Product.js`;
- Importeer React, schrijf een `Product` functie die een lege `<div>` teruggeeft en exporteer deze functie onderaan in het bestand. Let op: gebruik hiervoor een hoofdletter!
- Kopieer vervolgens alle HTML-elementen van het citroenen-artikel en zet dit op de plek van de lege `<div>`.

Als het goed is, ziet jouw code er nu zo uit:

```
// Product.js
import React from 'react';
function Product() {
  return (
    <article className="product">
      
      <h2 className="product-name">Citroen</h2>
      <p className="product-description">

```

```
        Een citroen is voor de meeste mensen te zuur om zo uit de hand te eten.  
        Van citroen kun je het vruchtvlees, het sap en de schil gebruiken. Het  
        sappige, lichtgele zure vruchtvlees versterkt de smaak van ander voedsel.  
    </p>  
    </article>  
);  
  
export default Product;
```

Om ons `<Product>`-component te gebruiken in ons `<App>`-component, moeten we dit component eerst importeren. We kunnen een component alleen importeren in een ander bestand, als het in diens eigen bestand geplaatst wordt. Als je nog even snel in `Product.js` kijkt, zie je dat we dat ook gedaan hebben:

- Importeer het `<Product>`-component in `App.js` door een `import statement` toe te voegen bovenaan het bestand. Tip: omdat het een default export is, hoeven we geen accolades te gebruiken.

Top! Dat werkt. Visueel is er niets veranderd, maar we gaan de code kant op. Ons component is alleen nog niet herbruikbaar... Als we meerdere `<Product>`-componenten neer zouden zetten, krijgen we talkens dezelfde content over citroenen. We moeten nu gaan werken met `properties`.

- Kijk eens goed naar de drie de attributen en benoem voor jezelf welke onderdelen verschillen. Welke property-namen zou je daarvoor bedenken?

- Gedoe deze properties (in de inhoud) mee aan het `<Product>`-component:

Je zou bijvoorbeeld kunnen kiezen voor `image`, `title` en `description`, en we geven de volgende data mee:

```
<main>  
    <Product  
        image={citroenen}  
        title="citroen"  
        description="Een citroen is voor de meeste mensen te zuur om zo uit de hand te eten. Van citroen kun je het vruchtvlees, het sap en de schil gebruiken. Het sappige, lichtgele zure vruchtvlees versterkt de smaak van ander voedsel."  
    />  
</main>
```

Side note: de `import-statements` voor de afbeeldingen blijven staan in `App.js`. Zo kunnen we de informatie vanuit `App.js` doorgeven aan de componenten en blijven ze herbruikbaar.

We geven deze eigenschappen nu mee, maar het `<Product>`-component weet nog niet wat het moet verwerken.

- Open het `Product.js`-bestand en voeg de parameter `props` toe aan de `<Product>`-functie.

- Nu kunnen we de bestaande `image-link` vervangen door onze `image-property`:

```
<img src={props.image} alt={props.title}/>
```

- ... en ten slotte kun je ook de `title` en de `beschrijving` vervangen. Ons component is nu volledig herbruikbaar!

- Vervang nu ook de andere twee HTML-stukken met het `<Product>`-component en geef de juiste properties voor limoenen en sjalotjes mee.

Zoi! Once webpagina is er! Je hebt nu geleerd hoe je standaard elementen en custom componenten gebruikt, hoe je properties doorgeeft, event listeners toevoegt en hoe je afbeeldingen en SVG's op een pagina weergeeft. Er zit echter nog weinig interactie in onze webpagina... In het volgende hoofdstuk leer je hier meer over.

Wil je checken of je alles goed hebt gedaan? Je kunt het eindresultaat van `App.js` [hier](#) bekijken. De uitleggingen voor het `Product`-component vind je [hier](#).

2.8 Speciale props doorgeven: children

Het doorgeven van properties is ontzettend handig, maar het kent ook limitaties. Zo zijn we altijd gebonden aan een vaste set properties. Heb jij jouw component zo gebouwd dat de property `title` wordt bepaald als `titel`? Dan kun je niet zomaar een afbeelding als `title` property meegeven. Logisch, maar soms hebben we meer vrijheid nodig.

Denk bijvoorbeeld aan het scenario waarin jouw interface opgebouwd is uit cards. Deze hebben altijd een titel, maar de "inhoud" is talkens anders. Denk aan tekst, afbeeldingen, of een combinatie daarvan.

React App

localhost:3000

Introductie

Lorem ipsum dolor sit amet, consectetur adipisicing elit. Deleniti explicabo hic molestiae provident saepe sunt, suscipit totam? Doloribus eos excepturi iure molestias, nulla voluptatem. Ad, cumque cupiditate dignissimos id magni maiores, officiis placeat possimus praesentium similique vel veniam voluptatem voluptatibus? Cum dolorem eligendi fugit illum, labore nostrum quibusdam rerum. Debitis, eligendi est iste mollitia nisi non odit pariatur perspiciatis placeat quam quasi reiciendis repellendus, tempore vero voluptas? Ab, ad alias, consectetur deleniti doloremque dolorum exercitationem explicabo fuga laudantium minus molestias necessitatibus neque non optio porro quidem quisquam! Architecto aspernatur corporis cupiditate fugit harum illum laborum, placeat, quidem rem sapiente ullam!

Sfeerimpressie



Quick actions

Aanmelden

Om dit te faciliteren heeft React een speciale property: `children`. Het zorgt ervoor dat we elementen kunnen weergeven tussen onze componenten. Stel dat je een card-component hebt gebouwd die de property `title` ontvangt. Die zou je dan toepassen als self-closing element:

```
function App() {
  return (
    <Card title="Quick actions"/>
  );
}
```

Echter, wanneer we ons component een opening- en closing tag geven, zullen alle elementen die we daarbinnen plaatsen (zoals titels, afbeeldingen, invoervelden, lijsten, etc.) worden gezien als children:

```
function App() {
  return (
    <Card title="Quick actions">
      <p>Dit nu een child van Card!</p>
      <p>Dit is ook een child van Card</p>
    </Card>
  );
}
```

Deze elementen kunnen we vervolgens via de children-property weergeven op de gewenste plek in ons component:

```
function Card(props) {  
  return (  
    <article>  
      <h1>{props.title}</h1>  
      {props.children}  
    </article>  
  );  
}
```

Deze veelzijdige property wordt in onderstaande video verder in detail toegelicht.

Wordt de video niet goed weergegeven of kun je deze niet in full-screen modus bekijken? Bekijk de video dan hier.

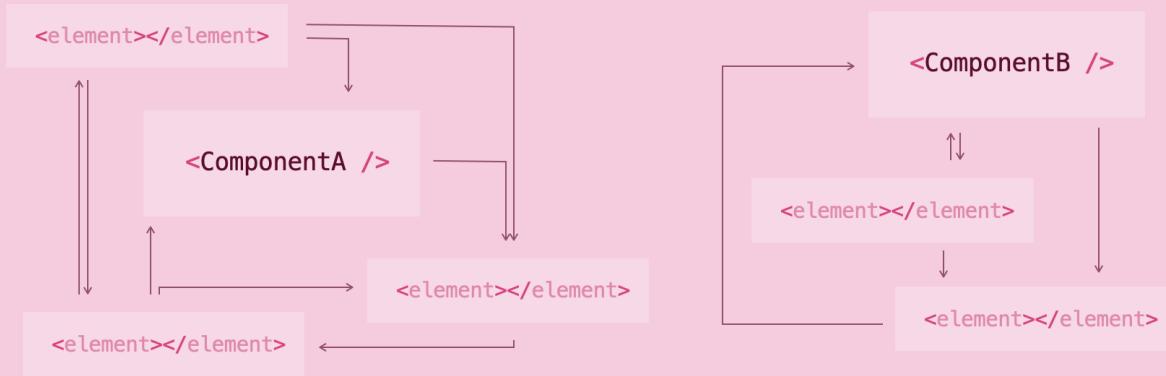
3. Interactieve componenten en formulieren

3.1 State

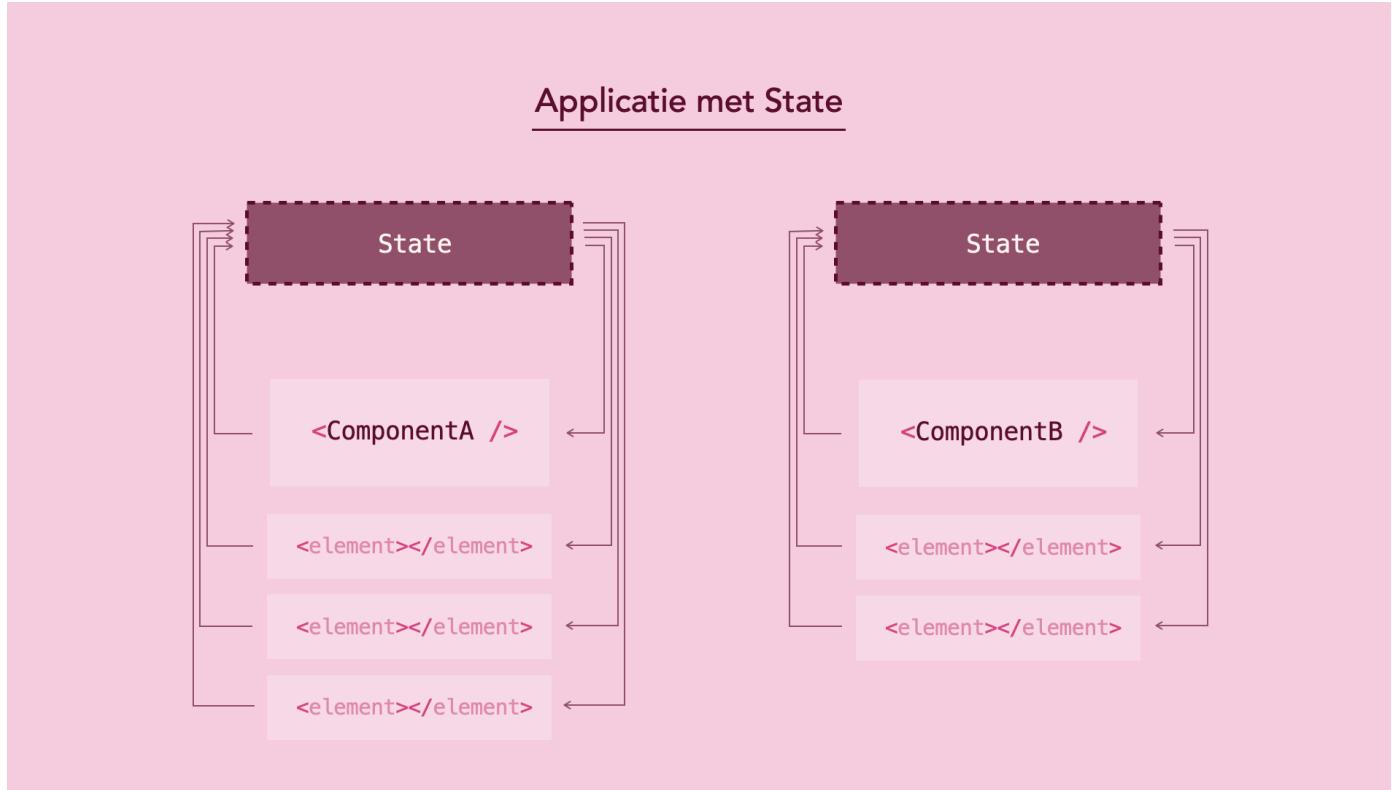
Tot nu toe hebben we alleen statische pagina's gemaakt met React: je weet nu hoe je werkt met afbeeldingen, teksten en andere componenten.

Met de schrijven van software - applicaties dus - draait alles om data. Niet alleen de data die wordt uitgewisseld tussen de backend en de frontend, maar ook de data die intern door onze React applicatie stroomt. We hebben informatie continu op verschillende plekken nodig en brengen hier wijzigingen aan. Wanneer je dit goed wilt organiseren, loop je al snel tegen een hindernis aan: we willen onze applicatie zo veel mogelijk opsplitten in verschillende componenten en features die verschillend taken uitvoeren.

Applicatie zonder State



Als we dit niet goed managen, is de data overal. Er is geen Single Source of Truth. Je kunt je voorstellen dat een applicatie op deze manier snel onoverzichtelijk wordt en dat onderhoud... nouja, een nachtmerrie is. Gelukkig kunnen we dit oplossen met State. Je kunt de State zien als een soort globale variabele waar alle elementen in ons component bij kunnen. De data staat op één plek en wordt alleen daarvandaan uitgelezen of gewijzigd.



Zie het als een gedachte kantoorruimte: je werkt daar het liefst op twee beeldschermen, maar niet met alle beeldschermen zijn voorzien van een HDMI-kabel. Zonder State zou je iedere ochtend een rondje moeten doen bij je collega's om te vragen of er ergens nog extra kabels liggen - en zo niet - of ze weten wie een extra kabel in gebruik heeft. Wanneer we State gebruiken, kun je iedere ochtend gewoon naar de balie lopen om een kabelje te vragen. De balie heeft de kabels in beheer en weet precies waar ze zijn, op elke moment van de dag, omdat alle medewerkers de kabels daar ophalen en weer leveren.

Zodra we onze applicatie van interactie willen voorzien, zullen we daar State voor gebruiken. In de State slaan we data op die veranderd, bijvoorbeeld de hoeveelheid items in een winkelmandje tijdens het online-shoppen, een lijst met zoekresultaten of de input van een wachtwoord-veld dat we bij iedere toetsaanslag willen valideren. Om waarden naar de State te schrijven en uit te lezen, gebruiken we speciale methodes genaamd **hooks**.

Hoe werkt dit precies? Stel dat we een spel spelen met meerdere spelers, waarbij de gebruikeremand de beurt mag geven. Door op een knop met de naam van de speler te klikken, zal deze speler aan de beurt komen. De actieve spelernaam willen we daaronder opslaan in de State.

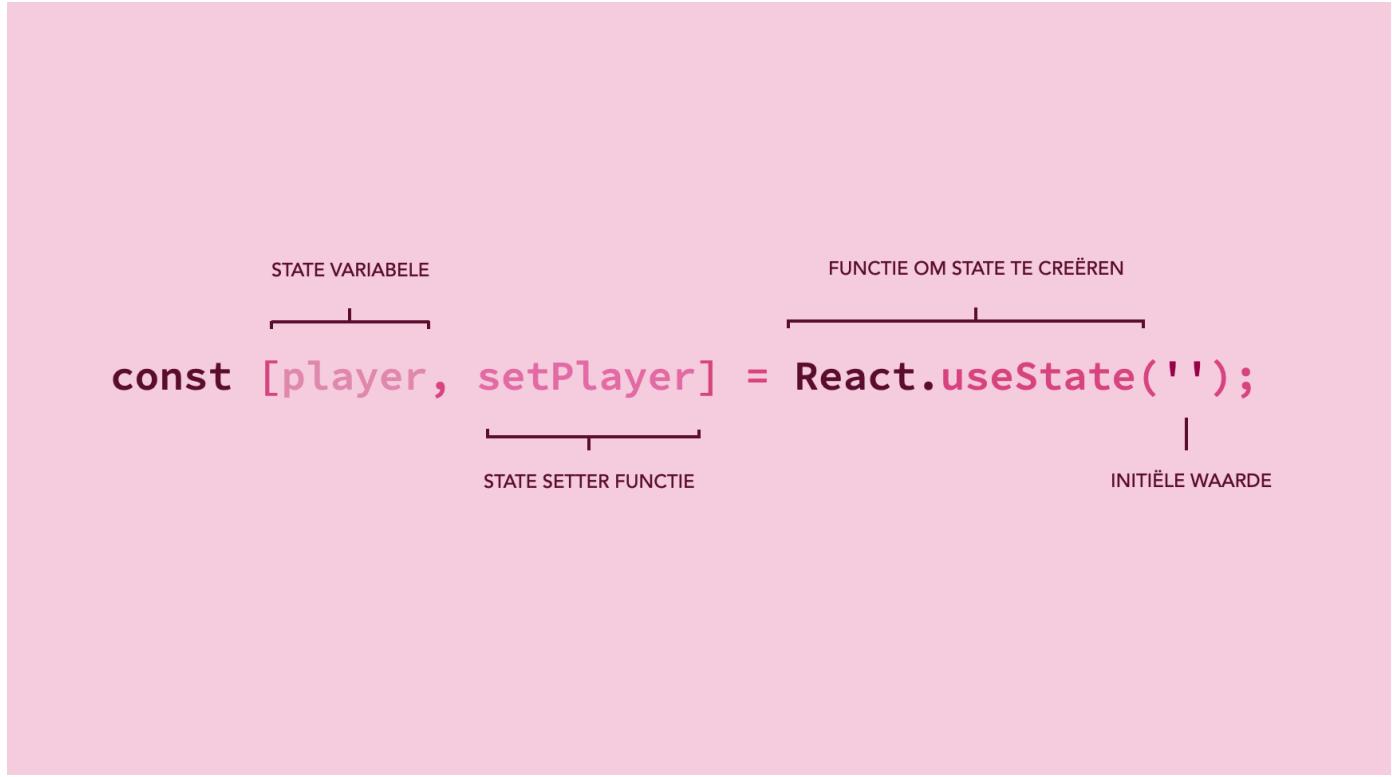
Marie is aan de beurt!

BART
PIET
MARIE

Om een plek in de State te creëren, gebruiken we de `useState`-hook:

```
const [player, setPlayer] = React.useState('');
```

Wanneer we deze methode aanroepen, geven we altijd een **initial waarde** mee: hiermee geven we aan met welke data we willen beginnen. Omdat we weten dat de speler-naam een string zal zijn, is het logisch om te beginnen met een lege string (' '). Indien je met getallen werkt, begin je vaak met 0 en als je met arrays gaat werken, begin je met een lege array ([]). Enzovoorts.



Inde keer als we de useState-methode aanroepen, krijgen we een array terug met twee waarden die we direct destructuren:

1. De eerste waarde is de key van het State object waaronder we onze informatie gaan opslaan. In ons geval kiezen we voor de naam `player`, maar je bent vrij om iedere naam te kiezen die je wilt.
2. De tweede waarde is de speciale setter-methode die gelinkt is aan de key, zodat we de gegevens die erin staan kunnen aanpassen. Ook deze naam mogen we zelf kiezen, hier noemen we de methode `setPlayer`.

zoals gebruikelijk is bij het destructuren van een array mag je de namen van hetgeen dat je terugkrijgt, zelf kiezen. Wanneer we een stukje State aannemken, zou dit ook mogen:

```
const [novi, softwareDevelopment] = React.useState('');
```

Het is echter totaal niet duidelijk dat `softwareDevelopment` een functie is en novi een key. En al helemaal niet dat ze bij elkaar horen! Het is daarom conventie om de bijbehorende methode altijd dezelfde naam te geven als de key, maar in een actieve vorm:

- setnaam van de State key of
 - toggleNaam van de State key
- Als je de key counter noemt, dan noem je de functie `setCounter` of `toggleCounter`. Wanneer je de key registered noemt, noem je de functie `toggleRegistered` of `setRegistered`. Vervolgens kun je overal in het component de naam van de key aanspreken en de setter-methode gebruiken om deze waarde aan te passen:

```
function App() {
  const [player, setPlayer] = React.useState('');

  return (
    <h1>{player} is aan de beurt</h1>
    <button type="button" onClick={() => setPlayer('Bart')}>Bart</button>
    <button type="button" onClick={() => setPlayer('Piet')}>Piet</button>
    <button type="button" onClick={() => setPlayer('Marie')}>Marie</button>
  );
}
```

In onderstaande video wordt het concept van State verder toegelicht aan de hand van het spelervoorbeeld.

Wordt de video niet goed weergegeven of kun je deze niet in full-screen modus bekijken? Bekijk de video dan [hier](#).

3.2 Condities

Eén van de dingen waar we state veel voor gebruiken, is het wisselen tussen weergave of styling van elementen. Zo kun je ervoor kiezen om foutmeldingen pas te tonen wanneer de gebruiker een regel overtreedt, of om een element een gekleurde rand te geven wanneer deze enkel is aangeklikt.

Conditioneel renderen

In het spel voorbeeld is er bij het opstarten van de applicatie nog niemand gekozen. Dat staat een beetje vreemd, want staat er bovenaan het scherm "... is aan de beurt". Het zou fijner zijn als we deze zin alleen tonen wanneer er daadwerkelijk iemand gekozen is en de `player`-variable daadwerkelijk een naam bevat. We willen het `h1`-element dus **conditioneel renderen**. Hiervoor zullen we de logica operatoren gebruiken die je eerder tijdens de cursus JavaScript hebt gebruikt: `if` (`&&`) en `if` (!) (zie "Basislissenstructuren" in JavaScript Hst. 2).

Dit doen we als volgt:

```
// IMPLICIETE CONDITIE:
// alleen renderen wanneer player een truthy waarde bevat
{player && <h1>(player) is aan de beurt</h1>}

// EXPLICiete CONDITIE:
// alleen renderen wanneer player NIET gelijk is aan een lege string
{player !== '' && <h1>(player) is aan de beurt</h1>}
```

Wanneer we data of condities tussen onze HTML-elementen (JSX) willen plaatsen, gebruiken we accolades () om dit te omwikkelen. In bovenstaand voorbeeld hebben we een conditie gedefinieerd (`player !== ''`), gevolgd door een logica operator: `&&` (en). Alleen wanneer de conditie true of truthy is, wordt het element na de `&&` weergegeven op de pagina. Wanneer de gebruiker vervolgens een button aanklikt verschijnt de titel er wordt dan aan de conditie voldaan.

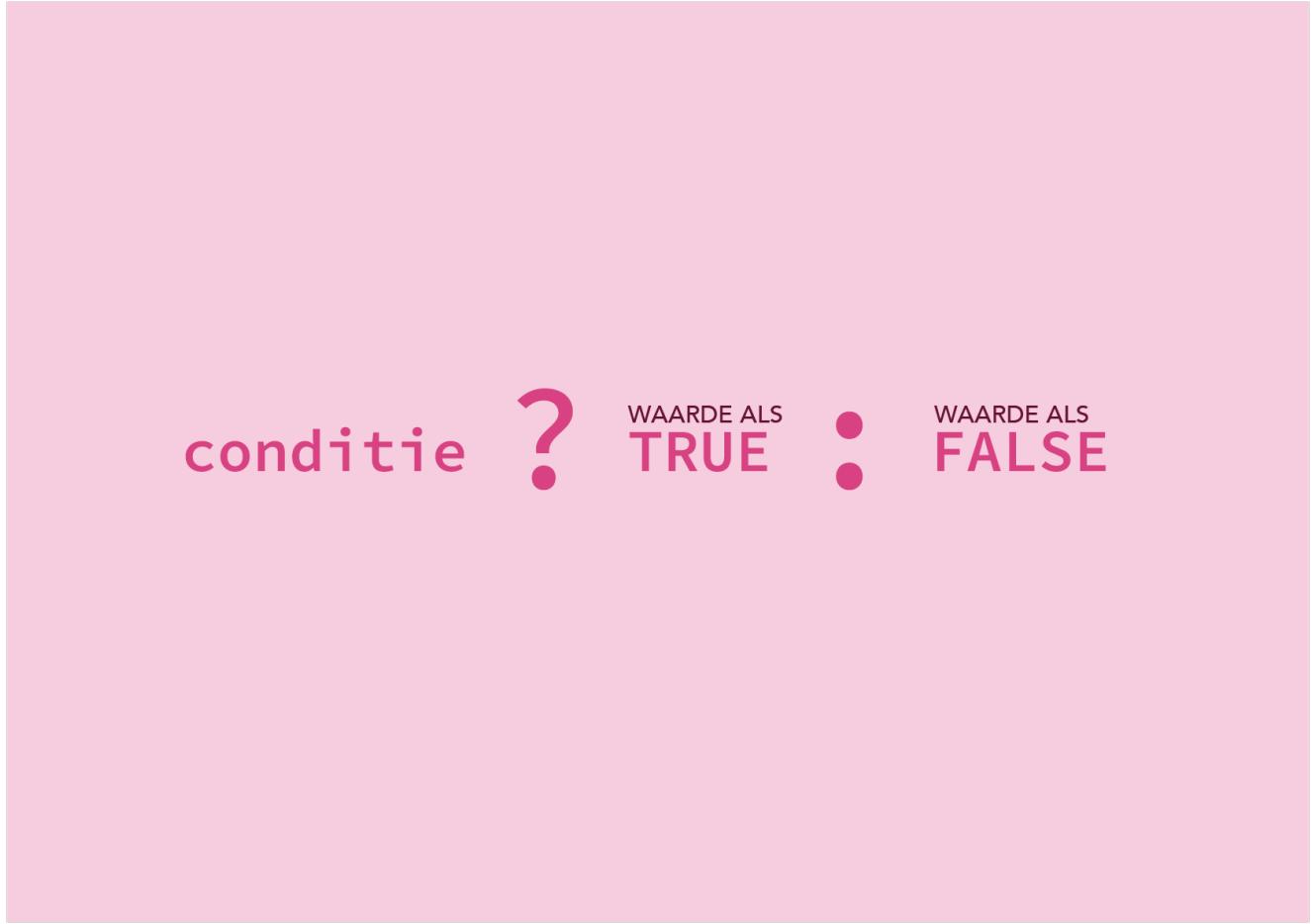
Maar wat als we de titel willen tonen wanneer er iemand gekozen is, maar een andere string willen tonen wanneer er niemand gekozen is? Ook hier hebben we een oplossing voor. Hier kunnen we een **ternary operator** (conditie ? true : false) gebruiken:

```
// Implicit gedefinieerd: bevat player een truthy waarde?
{player ? <h1>(player) is aan de beurt</h1> : <p>Er is nog niemand gekozen.</p>}

// Explicit gedefinieerd: player mag NIET gelijk zijn aan een lege string
{player !== '' ? <h1>(player) is aan de beurt</h1> : <p>Er is nog niemand gekozen.</p>}
```

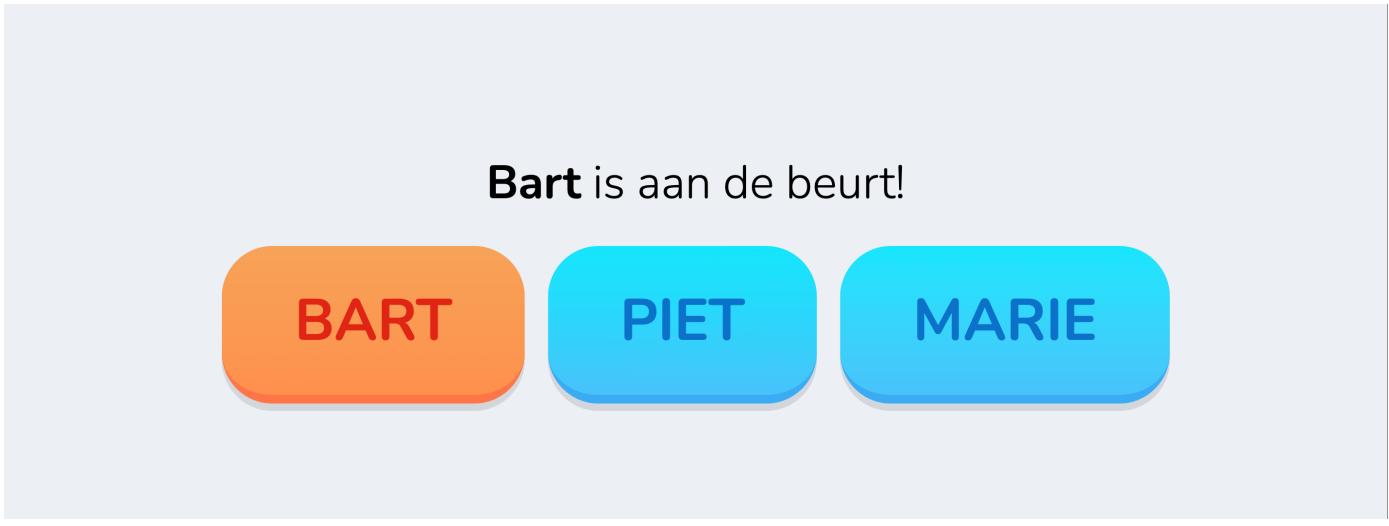
Een ternary operator werkt hetzelfde als een if-else-statement, maar is korter:

- Voor het ? staat er de conditie die getest wordt
- Als de conditie waar is, wordt de eerste waarde getoond
- Als de conditie niet waar is, wordt de tweede waarde (na de :) getoond



Conditioneel stylen

Net als dat we elementen conditioneel kunnen renderen, kunnen we ook CSS-classes toewijzen op basis van condities. Wanneer we bijvoorbeeld onze button een ander uiterlijk willen geven als deze speler op dit moment aan de beurt is:



Voor de ene versie hebben we de styling van css-class 'active' nodig en voor de andere versie de class 'default':

```
.active {  
  background-color: orange;  
}  
.default {  
  background-color: blue;  
}
```

Hier kunnen we weer een ternary operator voor gebruiken die de juiste class op het element plaatst:

```
<button  
  type="button"  
  onClick={() => setPlayer('Bart')}>  
  className={player === 'Bart' ? 'active' : 'default'}  
  > Bart  
</button>
```

Iedere keer als de State verandert, zal ook de conditie opnieuw gecontroleerd worden en wordt de class opnieuw op het element geplaatst.

Conditionele properties

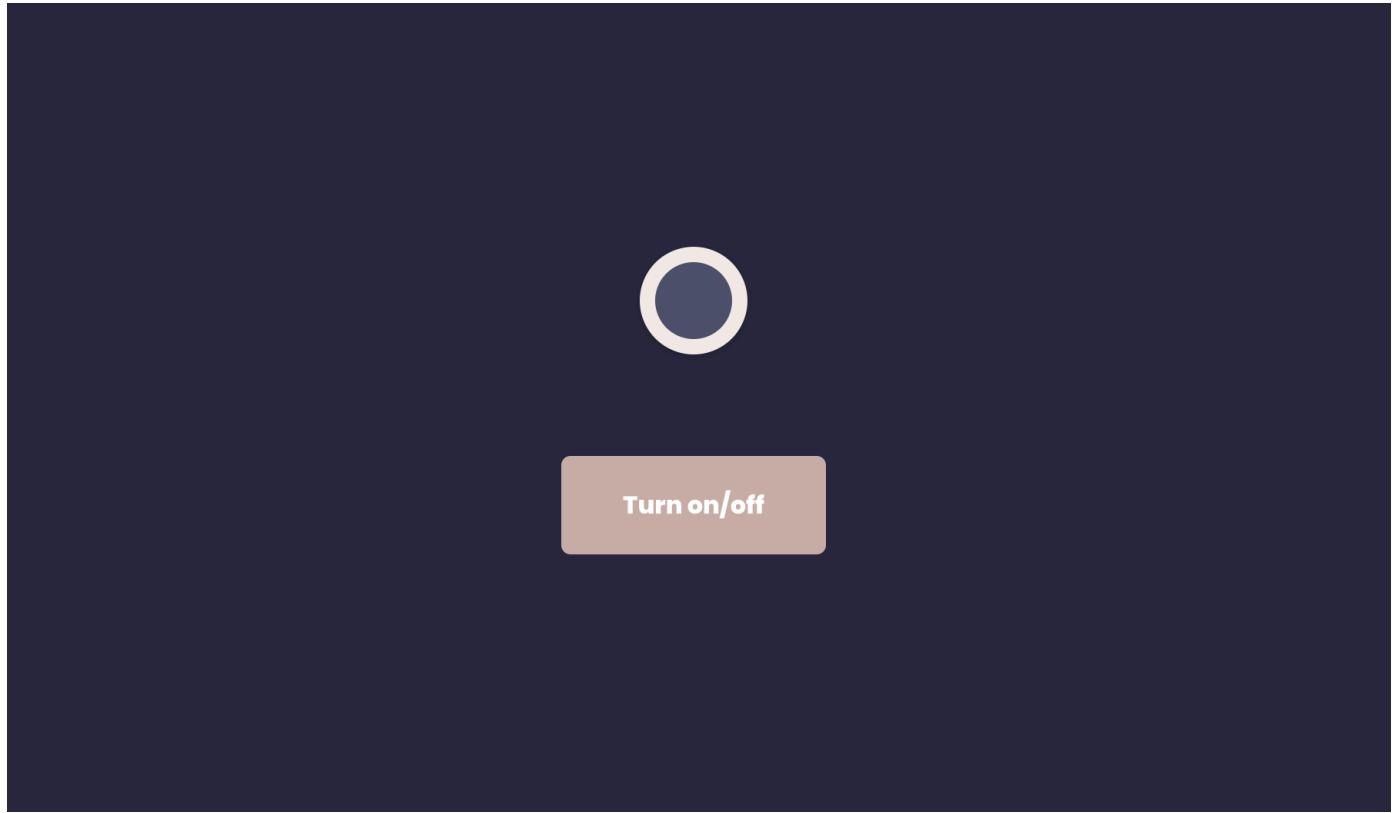
Op basis van dezezelfde state, zullen we wellicht ook andere properties of attributen willen doorgeven. Zo willen we de 'Bart'-button bijvoorbeeld disabled als Bart op dat moment al aan de beurt is. Het button-attribut disabled verwacht de boolean waarde true of false. Wanneer we hier een conditie aan meegeven, zal die conditie uiteindelijk evalueren naar één van die twee waarden:

```
<button
  type="button"
  onClick={() => setPlayer('Bart')}
  className={player === 'Bart' ? 'active' : 'default'}
  disabled={player === 'Bart'}
>
  Bart
</button>
```

In onderstaande video worden conditionele renderen en styles vader toegelicht.
Werd de video niet goed weergegeven of kun je deze niet in full-screen modus bekijken? Bekijk de video dan [hier](#).

3.3 Opdracht: zet het licht uit

Je gaat ervoor zorgen dat de gebruiker het licht van deze applicatie aan- en uit kan zetten. Dit ga je doen door gebruik te maken van State. Bovendien zal de gebruiker straks direct kunnen zien wanneer het licht aan of uit staat, door het gebruik van conditionele styling. Om de opdracht te maken kun je de opdracht klonen of downloaden naar jouw eigen computer via [diese GitHub repository](#). De uitwerkingen staan op de branch uitwerkingen.



Volg onderstaande stappen één voor één op:

1. Begin met het creeren van State, om daarin op te staan of het licht uit (false) of aan (true) staat. Belangrijk: geef de waarde van de state weer in een console.log, zodat je het kunt zien wanneer de waarde van de state verandert.
2. Zorg ervoor dat wanneer de gebruiker op de knop klikt, de State waarde wordt omgedraaid. Dit doe je door de state-setter methode te gebruiken die je in state heeft aangemaakt. Dus: is de waarde false? Dan wordt de waarde true? Dan wordt de waarde false. Is de waarde true? Dan wordt de waarde false. Hier voor zul je de `useState` moet gebruiken. Tip: test of het ook werkt als je vaker achtereen blijft klikken.
3. Voeg conditionele styling toe. Wanneer het licht aan staat, moet de class en op het main-element staan. Wanneer het licht uit staat, moet de class off op dit element staan.
4. Maak van beide de button-kleur dynamisch. Wanneer het licht aan staat, geef je de tekst 'Turn off' weer. Wanneer het licht uit staat, geef je de tekst 'Turn on' weer.

Applicatie starten

Als je het project gedownload hebt naar jouw lokale machine, installeer je eerst de `node_modules` door het volgende commando in de terminal te runnen:

```
npm install
```

Wanneer dit klaar is, kun je de applicatie starten met behulp van:

```
npm start
```

... of gebruik de WebStorm knop (from start). Open <http://localhost:3000> om de pagina in de browser te bekijken. Begin met het maken van wijzigingen in `src/App.js`, elke keer als je een bestand oplaat, zullen de wijzigingen te zien zijn op de webpagina.

3.4 Props vs. Callback props

In het vorige hoofdstuk heb je geleerd hoe je componenten properties kunt meegeven. Dit waren tot nu toe altijd primitive datatypes, zoals booleans, strings, getallen, arrays of objecten. In veel gevallen moeten we echter ook functies doorgeven aan onze componenten om ze echt herbruikbaar te maken. Denk aan de functie die moet worden uitgevoerd bij het klikken op een button-component, of de functie die nodig is om validate op een input-component toe te passen, iedere keer als de gebruiker een toetsaanslag maakt.

Wanneer we een functie als argument meegeven aan een andere functie, noem je dit een **callback**. Dit heb je al eerder gezien bij JavaScript, in bijvoorbeeld de map-methode: deze methode verwacht een functie als argument:

```
array.map(() => {
  // dit is de callback, waarin we telkens nieuw gedrag kunnen implementeren
});
```

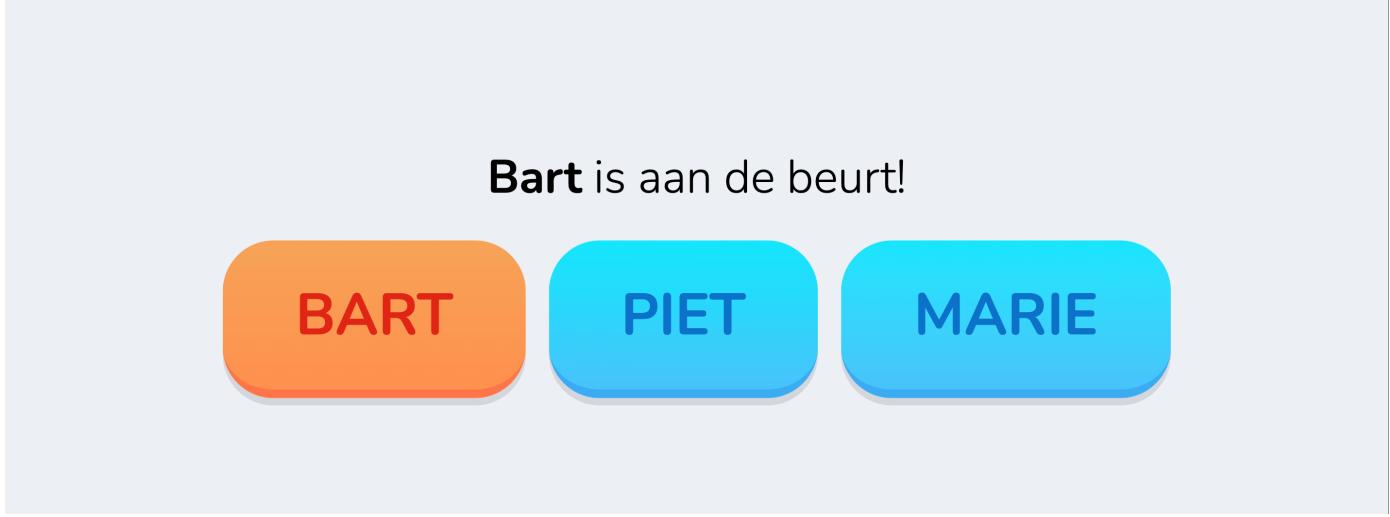
In bovenstaand voorbeeld gebruiken we toevallig een anonyme (ofwel: naamloze) arrow function `() => ()`. Maar wanneer we een gewone functie meegeven aan een andere functie, noem je dit ook een callback:

```
array.map(doThings);
function doThings() {
  // dit is de callback, waarin we telkens nieuw gedrag kunnen implementeren
}
```

Geef je een waarde mee aan je component? Dan noemen we dit een **prop**. Geef je een functie mee aan je component? Dan spreken we van een **callback prop**.

In beide gevallen geef je een functie mee: het enige verschil is de schrijfwijze. Stelkem heb je dus al heel veel met callbacks gewerkt! Wanneer we functies willen meegeven aan onze componenten, doen we dit op dezelfde manier als het doorgeven van normale properties. We noemen het dan echter geen properties meer, maar spreken van callback properties.

Maar hoe passen we dat toe? Laten we nog eens kijken naar onze spel-applicatie uit één van de vorige paragrafen:



```
function App() {
  const [player, setPlayer] = useState('');
  return (
    <>
      <h1>{player} is aan de beurt</h1>
      <button type="button" onClick={() => setPlayer('Bart')}>Bart</button>
      <button type="button" onClick={() => setPlayer('Piet')}>Piet</button>
      <button type="button" onClick={() => setPlayer('Marie')}>Marie</button>
    </>
  );
}
```

zoals je kunt zien, gebruikt iedere button de setPlayer-methode. Maar stel dat we van deze button-elementen, custom button-componenten willen maken. Welke properties moet dit component dan ontvangen?

...

Inderdaad, de naam van de speler en de methode om de state aan te kunnen passen. Alle buttons maken immers gebruik van dezelfde methode. Laten we deze als playerName (property) en handlePlayerChange (callback property) ontvangen in ons button-component:

```
// Button.js
function Button({ playerName, handlePlayerChange }) {
  return (
    <button
      type="button"
      onClick={() => handlePlayerChange(playerName)}
    >{playerName}</button>
  );
}

export default Button;
```

Zoals je ziet is er eigenlijk weinig veranderd ten opzichte van de eerdere situatie, alleen is "Bart" vervangen door playerName en de naam van de methode door handlePlayerChange. We roepen de functie nog steeds op dezelfde manier aan als je gewend bent.

```
// App.js
function App() {
  const [player, setPlayer] = useState('');
  return (
    <>
      <h1>{player} is aan de beurt</h1>
      <Button playerName="Bart" handlePlayerChange={setPlayer}/>
      <Button playerName="Piet" handlePlayerChange={setPlayer}/>
      <Button playerName="Marie" handlePlayerChange={setPlayer}/>
    </>
  );
}
```

Nu kunnen we dit component meerdere keren hergebruiken in App.js. We geven deze componenten telkens de naam van de speler en de state-setter-methode setPlayer mee. Let op: we gebruiken hier bewust geen ronde haken, omdat we de methode alleen doorgeven: we roepen de methode niet aan.

3.5 Controlled components

Toen je net begon met HTML, heb je je wellicht verbast over het gemak waarmee je invoervelden kon bouwen:

```
<input type="email" id="email-field"/>
```

Met één regel HTML had je een volledig functionerend wachtwoordveld: de gebruiker kan input inteksen, input plakken of deze weer weghalen en bovenind wordt de tekst niet weergegeven in karakters, maar in bolletjes! Al deze gratis functionaliteit krijg je van de DOM, die deze velden achter de schermen beheert. Wilde je de ingevulde waarden uitlezen met JavaScript? Dan deden we dat zo:

```
const emailField = document.getElementById('email-field');
console.log(emailField.value);
```

We noemen standard invoervelden daaronder **uncontrolled components**: wij doen niets en de DOM regelt alles. Wanneer we invoervelden in React bouwen, zetten we invoervelden (`useState`) op als **controlled components**: we overschrijven het standaard gedrag van de DOM met onze eigen logica, om zo de invoervelden te beheren met state. Zo kunnen we er bijvoorbeeld voor zorgen dat we invoervelden valideren tijdens het typen.

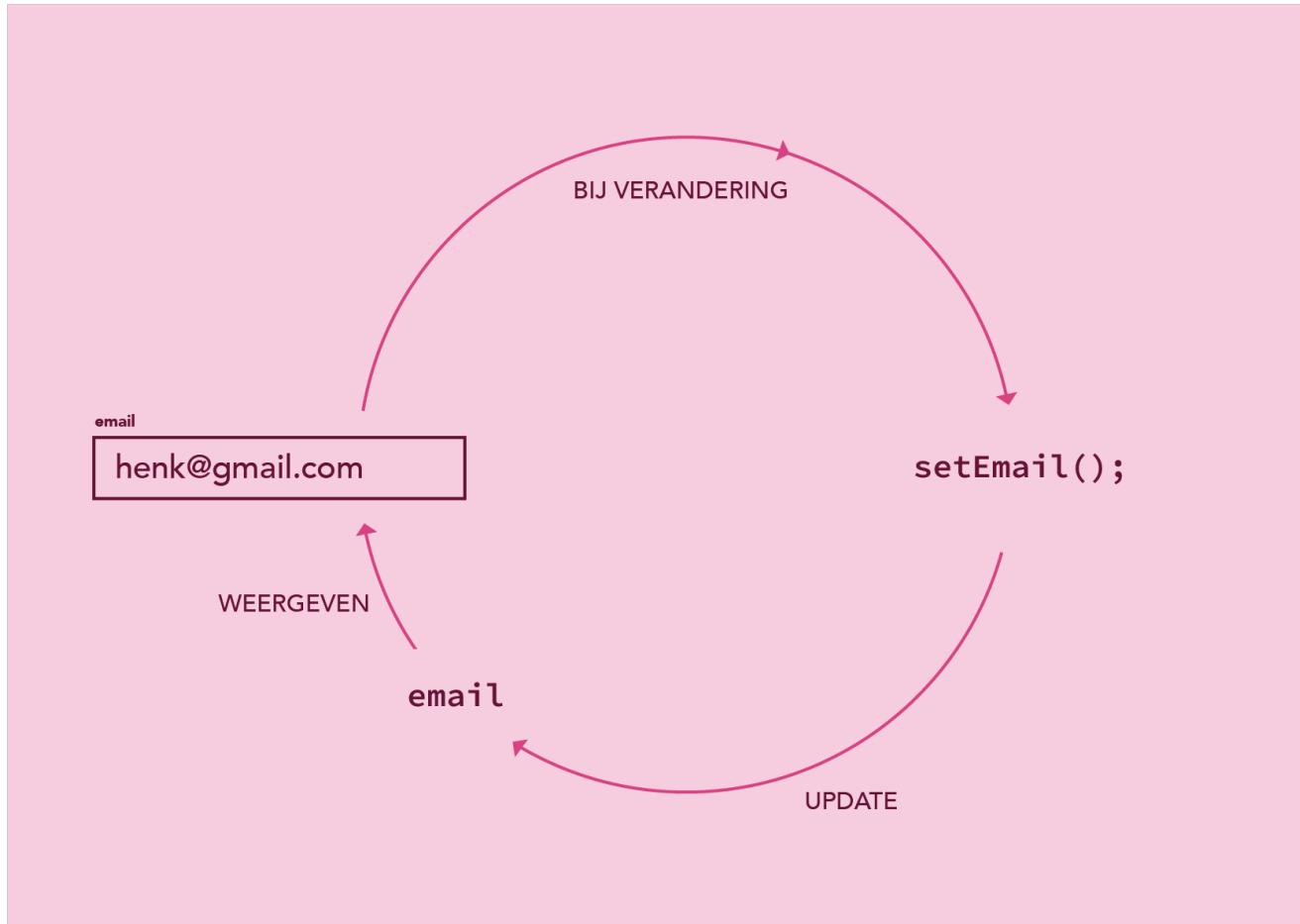
Maar hoe werkt dit? Eerst maken we een state waar voor ons email-veld:

```
function App() {
  const [emailValue, setEmailValue] = useState('');
  return (
    <input type="email"/>
  );
}
```

Vervolgens moeten we zorgen dat de inhoud van ons tekteveld gelijk staat aan de inhoud van de state. Namelijk, aan de state-variable `emailValue`:

```
<input type="email" value={emailValue}/>
```

Ten slotte willen we iedere keer als de gebruiker iets aan de waarde van het veld verandert, dit ook in de state wordt aangepast. Want die waarde gebruiken we vervolgens weer om de gebruiker te laten zien wat er is ingevoerd: het wordt een soort loop:



Om dit voor elkaar te krijgen, heeft ons inputveld een event listener nodig:

```
function App() {
  const [emailValue, setEmailValue] = useState('');
  return (
    <input
      type="email"
      value={emailValue}
      onChange={(e) => setEmailValue(e.target.value)}
    />
  );
}
```

Iedere keer wanneer de gebruiker een letter intikt, geeft onze event listener een Event-object terug. In dat Event-object zit heel veel informatie, waaronder de waarde van het veld (`e.target.value`). Die waarde gebruiken we als argument voor onze setter functie, zodat we de `emailValue` state variable kunnen updaten. Nu hebben we de waarde van het wachtwoordveld ten alle tijden beschikbaar in de State en kunnen alle andere elementen er altijd bij.

In onderstaande video wordt het implementeren van een controlled tekstveld en een controlled checkbox gedemonstreerd.

Werd de video niet goed weergegeven of kun je deze niet in full-screen modus bekijken? Bekijk de video dan [hier](#).

3.6 Opdracht: controlled components

Je gaat een React formulier maken met controlled components. Hiervoor maak je gebruik van de React `useState` hook. Je kunt de opdracht clonen of downloaden naar jouw eigen computer via [GitHub](#). GitHub repository. De uitwerkingen staan op de branch uitwerkingen en bonus-uitwerkingen.

Opdrachtdeskriptie

In jouw formulier komen de volgende inputs te staan:

- Naam - tekstveld
- Leeftijd - getalveld
- Opleidingen - selectieveld
- Checkvak voor de nieuwsbrief - checkbox
- Versturen-knop (van type submit)

Je gaat dit formulier bouwen door de instructies als controlled components op te zetten. Lees de volledige opdrachtdeskriptie in de `README.md` van de opdracht.

Applicatie starten

Als je het project gedownload hebt naar jouw lokale machine, installeer je eerst de `node_modules` door het volgende commando in de terminal te runnen:

```
npm install
```

Wanneer dit klaar is, kun je de applicatie starten met behulp van:

```
npm start
```

of gebruik de WebStorm knop (`npm start`). Open `http://localhost:3000` om de pagina in de browser te bekijken. Begin met het maken van wijzigingen in `src/App.js`: elke keer als je een bestand opslaat, zullen de wijzigingen te zien zijn op de webpagina.

3.7 Meerdere inputs met één `onChange`-handler

Zolang we één of twee invoervelden in ons formulier hebben staan, trekken onze state-initialisaties weinig aandacht. Maar een gemiddeld formulier waar je de gebruiker vraagt zijn adresgegevens in te vullen, telt zo al gauw zeven of acht invoervelden! Dat betekent dat onze componenten snel rommelig kunnen worden en zijn we veel `onChange`-properties aan het halen:

```
const [firstname, setFirstname] = useState('');
const [lastname, setLastname] = useState('');
const [age, setAge] = useState(0);
const [zipcode, setZipcode] = useState('');
const [city, setCity] = useState('');
const [remark, setRemark] = useState('');
const [agreedNewsletter, toggleAgreeNewsletter] = useState(false);



```

```
/>
```

Gelukkig is er ook een manier om één onChange handler te schrijven die dit afhandelt voor alle velden. Hier gaan we in deze paragraaf verder op in.

Een enkel input veld

Laten we eerst eens beginnen met het firstname-veld. Hiervoor zullen we, in plaats van een losse state variabele, één state-object voor initialiseren. Voor nu houden we het even bij de property `firstname` voor ons firstname veld, maar hier kunnen we straks heel gemakkelijk alle invoervelden aan toevoegen die we willen:

```
function App() {
  const [formState, setFormState] = useState({
    firstname: ''
  });

  return (
    // elementen...
  )
}
```

Let op: deze keys moeten overeenkomen met het name-attribuut van het desbetreffende invoerveld (<input name="firstname"/>).

Om ervoor te zorgen dat we zo min mogelijk code hoeven te schrijven, maken we een aparte handleChange functie die we kunnen aanroepen wanneer de gebruiker een toetsaanslag maakt. Deze kunnen we in de toekomst namelijk hergebruiken voor alle inputs in dit formulier. Omdat we nu nog maar één invoerveld hebben, wijzigen we telkens de `firstname`-key in het formState-object.

```
function App() {
  const [formState, setFormState] = useState({
    firstname: ''
  });

  function handleChange(e) {
    setFormState({
      firstname: e.target.value,
    });
  }

  return (
    <input
      type="text"
      name="firstname"
      value={formState.firstname}
      onChange={handleChange} // <- deze schrijfwijze is hetzelfde als (e) => handleChange(e). Maar omdat het event-object automatisch wordt doorgegeven, is enkel de functienaam voldoende
    />
  );
}
```

Iedere keer als de gebruiker iets intikt, wordt dit opgeslagen in de state. Om de loop nu compleet te maken, moeten we ervoor zorgen dat het value-attribuut altijd overeenkomt met de waarde uit de state:

```
<input
  type="text"
  name="firstname"
  value={formState.firstname}
  onChange={handleChange}
/>
```

Mooi, dit werkt voor ons voornaam-veld. Maar wat als we nog meer invoervelden willen toevoegen?

Meerdere input velden

Eerst zullen we een nieuwe key moeten toevoegen aan ons `FormState`-state-object. Uiteraard moet deze key weer overeenkomen met het `name`-attribuut van het nieuwe inputveld.

```
function App() {
  const [formState, setFormState] = useState({
    firstname: '',
    lastname: ''
  });

  function handleChange(e) {
    setFormState({
      firstname: e.target.value,
    });
  }

  return (
    <>
      <input
        type="text"
        name="firstname"
        value={formState.firstname}
        onChange={handleChange}
      />
      <input
        type="text"
        name="lastname"
        value={formState.lastname}
        onChange={handleChange}
      />
    </>
  );
}
```

We moeten alleen één belangrijke verandering aanbrengen aan onze `handleChange`-functie, zodat hij vanaf nu opgewassen is tegen oneindig veel extra invoervelden. Omdat we onze `handleChange`-functie aanroepen bij een toetsaanslag in zowel het achternaam- als voornaam-veld, moeten we weten welk invoerveld er zojuist veranderd is zodat we de juiste waarde in de state kunnen updaten.

Een fijne bijzonderheid is dat het event-object dat bij iedere toetsaanslag gegenereerd wordt, naast de ingevulde waarde ook de naam van het invoerveld registreert. Dit kunnen we terugvinden op `e.target.name`.

Naam invoerveld: e.target.name Waarde invoerveld: e.target.value

Dit betekent dat wanneer we de letter `N` intoetsen in het achternaam-veld, we dit zo kunnen terugvinden:

```
function handleChange(e) {
  console.log(e.target.name); // geeft 'lastname'
  console.log(e.target.value); // geeft 'N'
}
```

Die `N` willen we opslaan in de key `lastname` van het `FormState`-object. Maar hoe doen we dit? Dit mag namelijk niet, gezien een objectKey geen punten mag bevatten:

```
function handleChange(e) {
  setFormState({
    e.target.name
  : e.target.value, // de applicatie crasht
  });
}
```

"Oh joh, geen probleem." - denk je nu. Dan zetten we 'm toch gewoon in een variabele?

```
function handleChange(e) {
  const changedFieldName = e.target.name;
  setFormState({
    changedFieldName: e.target.value,
  });
}
```

Nou - leuk idee! - maar in plaats van een key genaamd `lastname`, hebben we nu letterlijk een `changedFieldName`-key in ons object gezet..

Elements Console Sources Network > 1 Default levels 1 Issue: 1

▼ Object i
changedFieldName: "N"
► [[Prototype]]: Object

App.js:23

> |

* Wat je zou zien als je de state zou loggen binnen de return statement

We zullen het dus anders moeten oplossen, maar gelukkig zijn we wel erg in de buurt. Wanneer je een variabele (zoals e.target.value of changedFieldName) als object-key wil gebruiken, gebruiken we de alternatieve blokhaak syntax:

```
function handleChange(e) {
  const changedFieldName = e.target.name;
  setFormState({
    [changedFieldName]: e.target.value,
  });
}
```

Op deze manier wordt de inhoud van de variabele als key gebruikt, niet de naam van de variabele zelf. Maarrrr, we zijn nog niet klaar. Wat we nu doen, is de huidige state overschrijven met een object dat alleen de key bevat van het veld dat zojuist aangepast is. We zijn de data uit onze andere invoervelden nu kwijt: die hebben we overschreven...

Hoe zorgen we er voor dat de huidige waarden van het object bewaard blijven, maar we alleen de `keyname`-key overschrijven? Dat doen we met de `spread operator`. De spread operator (...) kopieert simpelweg de properties van een bestaand object naar een nieuw object. Alleen de key die we ondernaam toevoegen, wordt overschreven:

```
function handleChange(e) {
  const changedFieldName = e.target.name;
  setFormState({
    ...formState,
    [changedFieldName]: e.target.value,
  });
}
```

Selectbox en/of Textarea

Het gebruik van select- en textarea-elementen werkt op dezelfde manier zoals we eerder hebben gezien bij input-elementen. We voegen een key toe aan ons state object, plaatsen de `handleChange`-functie op het element en geven de juiste value-property mee. Onze `handleChange`-functie is al herbruikbaar en hoeft niet aangepast te worden.

```
function App() {
  const [formState, setFormState] = useState({
    firstname: '',
    lastname: '',
    gender: 'neutral',
  });

  /* handleChange functie */

  return (
    <select
      name="gender"
      value={formState.gender}
      onChange={handleChange}>
      <option value="neutral">Neutral</option>
      <option value="male">Male</option>
      <option value="female">Female</option>
    </select>
  );
}
```

Checkboxes en radio buttons

Radio buttons werken een klein beetje anders dan reguliere form-elementen. Omdat radio-buttons "single-selected" opties zijn (slechts één antwoord mogelijk), moeten alle name-attributen van radio-inputs die onderdeel zijn van dezelfde vraag, identiek zijn. Daarnaast geven we met het checked-attribuut aan welke optie op dit moment geselecteerd is:

```
function App() {
  const [formState, setFormState] = useState({
    firstname: '',
    lastname: '',
    gender: 'neutral',
    moment: 'night',
  });

  /* handleChange functie */

  return (
    <>
      <label>
        Nacht
        <input
          type="radio"
          name="moment"
          value="night"
          checked={formState.moment === "night"}
          onChange={handleChange}
        />
      </label>
      <label>
        Dag
        <input
          type="radio"
          name="moment"
          value="day"
          checked={formState.moment === "day"}
          onChange={handleChange}
        />
      </label>
    </>
  );
}
```

Om checkboxes volledig te kunnen gebruiken moeten we een kleine aanpassing maken in onze `handleChange`-functie. Het checked-attribuut mag hier namelijk alleen een true of false waarde bevatten en deze waarde vinden we niet op `e.target.value`, maar op `e.target.checked`. We zullen dus in onze `handleChange`-functie moeten controleren welk type invoer-veld op dat moment getypt wordt - en als dat een checkbox is, kiezen we voor de checked waarde:

```
function App() {
  const [formState, setFormState] = useState({
    firstname: '',
    lastname: '',
    gender: 'neutral',
    moment: 'night',
    conditions: false,
  });

  function handleChange(event) {
    const changedFieldName = event.target.name;
    const newValue = event.target.type === "checkbox" ? event.target.checked : event.target.value;

    setFormState({
      ...formState,
      [changedFieldName]: newValue,
    });
  }

  return (
    <label>
      <input
        type="checkbox"
        name="conditions"
        checked={formState.conditions}
        onChange={handleChange}
      />
      Akkoord met de algemene voorwaarden
    </label>
  );
}
```

)

Je weet nu hoe je alle soorten inputs kunt verwerken in één functie. Om dit voor elkaar te krijgen, zorg je dat je de name-attributen altijd overeenkomen met de namen in het state-object. Zo kunnen we de juiste key in de state overschrijven met de ingebetoste waarde, en raken we alle andere ingevulde gegevens niet kwijt door gebruik te maken van de spread-operator.

4. Bonus: React Hook Form

4.1 Inleiding

Je had inmiddels geleerd hoe je een controlled component maakt. En dat werkt prima, zolang je één of twee inputvelden in je formulier hebt staan. Maar zodra dit aantal stijgt naar zes of acht inputvelden, moet je eerst langs een gigantisch blok met useState() declaraties scrollen voor je bij de rest van je code komt. Uiteraard valt dit op te lossen door door slechts één state object en één event handler te gebruiken, zoals beschreven in Hiert 3.8. Maar wanneer je form-validate op al deze velden wil gaan toepassen, of complexere formulieren wil gaan bouwen, blijft deze oplossing vrij arbeidsintensief. Bovendien vraagt een controlled component ook heel veel processorkracht: het inputveld wordt bij eledere toetsaanslag opnieuw gerenderd.

Tja, en je raadt het al: dit vinden we in programmeertalend hartstikke inefficiënt.

Wanneer je grote, complexe formulieren nodig hebt in je applicatie is het daarom raadzaam om een library te gebruiken. Je hebt hier verschillende partijen voor, zoals [Formik](#), [Redux Form](#) of [React Hook Form](#).

React Hook Form is de meest compacte en lichtgewicht library van de drie, die we gemakkelijk in ons project kunnen installeren via npm. Het biedt een vereenvoudigde manier voor het maken van formulieren, waardoor het gemakkelijker is om validateregels toe te passen en we minder code hoeven te schrijven. Bovendien minimaliseert React Hook Form het aantal re-renders van het formulier. Dit zorgt dus ook nog eens voor snellere laadtijden!

“Maar hoezo heb ik dan net geleerd hoe ik een formulier maak met controlled components? – denk je nu wellicht. Nou, omdat het heel belangrijk is dat je weet hoe je met de basis-functionaliteit van React een formulier opbouwt, voor je aan de slag gaat met libraries en plugins. Vaak heb je een toepassing als React Hook Form ook pas nodig bij grotere implementaties. Wanneer je de koksopleiding doet ga je ook niet beginnen aan het maken van eleraalde als je nog niet weet hoe je een ei kokt, toch?

4.2 Invoervelden registeren

Laten we React Hook Form gaan implementeren in een formulier. We beginnen met onderstaand voorbeeld: een statisch formulier met twee text-inputs en een text-area:

```
function App() {
  return (
    <form>
      <label htmlFor="name-field">
        Naam:
        <input
          type="text"
          name="name"
          id="name-field"
        />
      </label>

      <label htmlFor="email-field">
        Email:
        <input
          type="text"
          name="email"
          id="email-field"
        />
      </label>

      <label htmlFor="message-field">
        Bericht:
        <textarea
          id="message-field"
          rows="4"
          cols="40"
          placeholder="Laat je bericht achter"
          name="message-content"
        ></textarea>
      </label>

      <button type="submit">
        Versturen
      </button>
    </form>
  );
}
```

Installatie

Om React Hook Form te implementeren, zullen we het eerst als dependency moeten installeren in ons project. Dit doe je door het volgende commando in de terminal te typen:

```
npm install react-hook-form
```

Wanneer dit klaar is, kunnen we useForm-methode van React Hook Form bovenaan ons App.js component importeren:

```
import { useForm } from 'react-hook-form';

function App() {
  return (
    /* formulier... */
  );
}
```

Tip: Omdat je niet kunt ruiken hoe een npm package zijn functies of componenten exposeert, kijk je altijd eerst even in de [documentatie](#)!

De useForm-methode geeft toegang tot alle functionaliteiten die deze veelzijdige package ons biedt. Want als je deze methode aan zou roepen, krijg je een object met functies terug. Om je een beter beeld te geven, hebben we dit voor je in de console gelogd:

Elements Console **Console** Sources Network » **1**

top ▾ Filter Default levels ▾

1 Issue: **1**

App.js:7

▼ Object

- **clearErrors**: *name => {}*
- **control**: *{register: f, unregister: f, getFieldState: f, _executeS...*
- **formState**: *{...}*
- **getFieldState**: *(name, formState) => {...}*
- **getValues**: *fieldNames => {...}*
- **handleSubmit**: *(onValid, onInvalid) => {...}*
- **register**: *f (name)*
- **reset**: *(formValues, keepStateOptions) => {...}*
- **resetField**: *f (name)*
- **setError**: *(name, error, options) => {...}*
- **setFocus**: *f (name)*
- **setValue**: *f (name, value)*
- **trigger**: *async f (name)*
- **unregister**: *f (name)*
- **watch**: *(name, defaultValue) => {...}*
- [[Prototype]]: Object

>

Registratie

Elke van de methodes die je altijd nodig hebt, is register. We moeten invoerden die we willen gaan beheren met React Hook Form namelijk altijd eerst registreren. Daarom is het raadzaam deze methode direct te destructuren uit de useForm-aanroep:

```
function App() {
  const { register } = useForm();
  return (
    /* formulier... */
  );
}
```

We gebruiken deze methode om elke input-element afzonderlijk aan te melden. Het argument dat we aan de register-methode meegeven, is de naam van het invoerveld. Dit zorgt ervoor dat we het name-attribuut niet meer apart als attribuut op ons element hoeven te plaatsen, doet deze methode nu voor ons. De register-methode plaatst dus zowel een referente, als een naam op het element. Dat zijn dus meerdere attributen in één, waarvoor we de spread operator (...) moeten gebruiken:

```
// Voor
<input
  type="text"
  name="email"
/>

// NA
<input
  type="text"
  {...register("email")}
/>
```

Bovenaan input-element is nu onderdeel van het React Hook Form, onder de naam email. En meer hoeven we niet te doen!

Daarmee wordt het hele verhaal met onChange-listeners en value-attributen in één keer voor ons geregeld. Je merkt het al: dit scheelt dus ontzettend veel werk in verhouding tot onze controlled components. Als we dit implementeren voor de rest van de invoervelden, ziet dit er zo uit:

```
function App() {
  const { register } = useForm();
  return (
    <form>
      <label htmlFor="name-field">
        Naam:
        <input
          type="text"
          id="name-field"
          {...register("name")}
        />
      </label>

      <label htmlFor="email-field">
        Email:
        <input
          type="text"
          id="email-field"
          {...register("email")}
        />
      </label>

      <label htmlFor="message-field">
        Bericht:
        <textarea
          id="message-field"
          rows="3"
          cols="40"
          placeholder="Laat je bericht achter"
          {...register("message-content")}
        />
      </label>
    </form>
  );
}
```

```

        >
      </textare>
    </label>
    <button type="submit">
      Versturen
    </button>
  </form>
)
);
}

```

In de volgende paragrafen leer je meer over submitten, validate en andere handige toepassingen.

4.3 Submitten

Een formulier moet uiteindelijk verzonden worden. Op de traditionele manier, zonder React Hook Form, zouden we dat als volgt implementeren:

```

function App() {
  function handleFormSubmit(e) {
    e.preventDefault();
    console.log('Hier willen we nu alle state waarden verzamelen!');
  }

  return (
    <form onSubmit={handleFormSubmit}>
      /* input velden... */
    </form>
  )
}

```

Op de <form>-tag staat een onsubmit-listener die wordt triggered zodra er op een submit-button binnen de <form>-tag wordt geklikt, of wanneer de gebruiker op Enter drukt. De functie die dan wordt uitgevoerd, hebben we handleFormSubmit genoemd. Die zorgt er met preventDefault() voor dat de pagina niet verstuurd wordt.

Maar hoe doen we dat in een React Hook Form? Om te beginnen hebben we de handleSubmit-methode nodig uit de useForm-aanroep:

```
const { register, handleSubmit } = useForm();
```

Vervolgens kunnen we deze methode om onze eigen handleFormSubmit-functie wikkelen:

```

return (
  <form onSubmit={handleSubmit(handleFormSubmit)}>
    /* input velden... */
  </form>
)

```

Voor de oplettende lezer: we geven onze eigen onFormSubmit functie dus eigenlijk mee als callback aan de handleSubmit-methode!

Onze onFormSubmit-functie werkt nu nog steeds, maar wordt al helemaal met React Hook Form functionaliteit. Zo hoeven we bijvoorbeeld geen e.preventDefault meer te gebruiken. Maar het zorgt er ook voor dat we in plaats van het Event-object, nu altijd een data-object van React Hook Form mee krijgen als parameter. Daarin heeft React Hook Form alle ingevulde waarden al voor ons verzameld:

```

function App() {
  const { register, handleSubmit } = useForm();

  function handleFormSubmit(data) {
    console.log(data);
  }

  return (
    <form onSubmit={handleSubmit(handleFormSubmit)}>
      /* input velden... */
    </form>
  )
}

```

Wanneer we nu op de verzend-knop drukken, zullen we het volgende in de console zien verschijnen:

```
{
  name: "Henk Pietersen",
  email: "henkpietersen@novi.nl",
  messageContent: "Ik wil graag een offerte ontvangen!"
}
```

De keys in het data-object komen dus overeen met de namen die we geregistreerd hebben met de register-methode. Mis je er eenbij? Dan ben je dat inwendig waarschijnlijk vergeten te registreren.

4.4 Validate regels

Één van de voordelen van React Hook Form is dat het valideren van de velden ontzettend makkelijk te implementeren is. Al het zware werk is al voor ons gedaan. Stel dat we het invoerveld voor het bericht van de gebruiker verplicht willen maken, maar ook willen afleiden dat er een minimaal- en maximaal aantal karakters gebruikt wordt. Om deze validatie toe te voegen, geven we onze register-functie simpelweg een tweede argument mee: een object met validatieregels:

```

<textarea
  {...register("messageContent", {
    required: true,
    minLength: 10,
    maxLength: 50,
  })}
></textarea>
```

Bovenaan staan zorg ervoor dat de gebruiker dit veld verplicht moet invullen, en dat de inhoud minimaal 10 en maximaal 50 karakters mag bevatten. Deze drie regels worden gecontroleerd zodra de gebruiker op "verzenden" drukt. De gebruikte keys, required, minLength en maxLength, hebben we niet zelf bedacht. Er is een standaard set validatieregels beschikbaar die je kunt meegeven, waaronder:

- Required - voor verplichte velden (afgedrukt in true of false);
- MinLength - voor een minimaal aantal karakters (in cijfers);
- MaxLength - voor een maximaal aantal karakters (in cijfers);
- Min - voor de minimale hoogte van getallen (in cijfers);
- Max - voor de maximale hoogte van getallen (in cijfers);
- Validate - voor zelfgemaakte validatieregels, zoals checken of het woord "pannenkoeken" voorkomt in de input (als callback functie);
- Pattern - voor het herkennen van een patroon, zoals een postcode (getal-getal-getal-letter-letter) of gebruik van speciale tekens (als RegEx patroon);

We kunnen nu de code van de register-functie gaan aanpassen. We gaan de required-regel toevoegen en de minLength-, maxLength- en maxmaal-regels verwijderen. We kunnen de maxmaal-regel verwijderen omdat de gebruiker nu alleen maar een e-mail-adres kan invullen. We kunnen de minLength-regel verwijderen omdat de gebruiker nu alleen maar een e-mail-adres kan invullen.

```

// Als je hier een aparte functie voor zou schrijven, zou je dat zo doen:
// function doesItIncludeAt(value) {
//   if (value.includes('@')) {
//     return true;
//   } else {
//     return false;
//   }
// }

// gezien de ingebouwde methode - includes - uit zichzelf al true of false teruggeeft,
// kan je dit zelfs verkorten tot:
function doesItIncludeAt(value) {
  return value.includes('@');
}

// gezien onze doesItIncludeAt-functie maar één regel code bevat,
// kunnen we dit als arrow-function zelfs verkorten tot:
// (value) => value.includes('@');
```

We kunnen de opzet van deze functie vervolgens gebruiken voor de validate-regel van React Hook Form. Deze regel verwacht een (callback) functie die een true of false waarde teruggeeft:

```

// Het e-mail veld is verplicht én moet een @ bevatten
<input
  type="text"
  {...register("email", {
    required: true,
    validate: (value) => value.includes('@'),
  })}
/>
```

Side note: het is professioneler om het bovenstaande voorbeeld op te lossen met een pattern. Een pattern wordt altijd opgebouwd door middel van regular expressions (RegEx) en wordt niet behandeld in dit curriculum. Als je wel benieuwd bent naar RegEx en hier meer over wil weten, zijn hier genoeg tutorials over te vinden, zoals bijvoorbeeld [deze](#) .

Foutmeldingen toevoegen

Goed, we hebben nu validatieregels opgesteld, maar de gebruiker tast nog in het duister wanneer het formulier niet verzonden wordt, omdat er niet aan de regels wordt voldaan. We moeten het overbrengen van deze regels daarom zo helder mogelijk op de gebruiker overbrengen. Omdat er soms meerdere validatieregels op één veld kunnen staan, is het handig om altijd direct een foutmelding toe te voegen waarin we beschrijven wat de gebruiker verkeerd doet. Dit bericht kunnen we dan later toepassen op de specifieke regel overbrenging.

We gaan de regels op ons textarea-element daarom uitbreiden. We waren begonnen met het onderstaande:

```

<textarea
  {...register("messageContent", {
    required: true,
    minLength: 10,
    maxLength: 50,
  })}
></textarea>
```

Om foutmeldingen toe te kunnen voegen, geven we ieder regel een eigen object:

```
<textarea
  {...register("message-content", {
    required: {},
    minLength: {},
    maxLength: {}
})}
></textarea>
```

Ieder object bevat op diens beurt weer een `value`- (de regel) en een `message`- (de foutmelding) key:

```
<textarea
  {...register("message-content", {
    required: {
      value: true,
      message: 'Dit veld is verplicht',
    },
    minLength: {
      value: 10,
      message: 'Input moet minstens 10 karakters bevatten',
    },
    maxLength: {
      value: 50,
      message: 'Input mag maximaal 50 karakters bevatten',
    }
})}
></textarea>
```

React Hook Form zorgt er tevens voor dat de validatoren `minLength` en `maxLength` pas gecontroleerd worden nadat er is voldaan aan de `required`-regel. Super handig! Wanneer je een foutmelding wilt instellen voor de `validate`-regel, werkt dit net ietsje anders. Dan zul je een `||`-operator (`|||`) moeten gebruiken:

```
<input
  type="text"
  {...register("email", {
    required: {
      value: true,
      message: 'Dit veld is verplicht',
    },
    validate: (value) => value.includes('@') || 'Email moet een @ bevatten',
  })}
/>
```

Foutmeldingen weergeven in de UI

De laatste stap is nu het weergeven van de foutmeldingen, zodat de gebruiker ze kan zien. Het eerste dat we daarvoor nodig hebben, is het `errors`-object van React Hook Form. Ook die halen we uit de `useForm`-call (het blijft maar komen):

```
const { handleSubmit, formState: { errors }, register } = useForm();
```

Wanneer de gebruiker ons naam-veld leeg zou laten en te veel karakters gebruikt in het opmerkingenveld, ziet ons `errors`-object er zo uit:

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. At the top, there are buttons for Elements, Sources, Network, and other developer tools. Below the tabs is a toolbar with icons for play/pause, stop, and refresh, followed by a 'Filter' input field and a 'Default levels' dropdown. The main area displays two error objects:

- App.js:27**:
`▼ {name: {...}, message-content: {...}} ⓘ`
`▼ message-content:
 ▶ message: "Input mag maximaal 50 karakters bevatten"
 ▶ ref: textarea#message-field
 ▶ type: "maxLength"
 ▶ [[Prototype]]: Object`
`▼ name:
 ▶ message: "Naam is verplicht"
 ▶ ref: input#name-field
 ▶ type: "required"
 ▶ [[Prototype]]: Object
 ▶ [[Prototype]]: Object`
- App.js:27**:
`>`

Zoals je ziet, is er een key voor ieder veld dat op dat moment niet voldoet aan de validatorenregels. De velden die goedkeurd zijn, staan namelijk `ref` in dit error-object. Dit zorgt ervoor dat we op basis van de aanwezigheid van de `key`, weten of we de melding moeten tonen. Dit doen we door de melding conditioneel te renderen:

```
<input
  type="text"
  id="name-field"
  {...register("name", {
    required: {
      value: true,
      message: 'Naam is verplicht'
    }
})}
/>
{errors.name && <p>{errors.name.message}</p>}
```

We checken of er een `name`-key op het object staat - en als dat zo is - geven we de bijbehorende foutmelding weer. Heb je een veld geregistreerd waar een streepje tussen staat, zoals we bij de `textarea` hebben gedaan? Dan zul je de `blokhaak notatie` in plaats van de `punt-notatie` moeten gebruiken om de juiste key aan te spreken:

```
{errors['message-content'] && <p>{errors['message-content'].message}</p>}
```

Wil je deze theorie eerst nog even in actie zien? Onderstaande video neemt je stap voor stap mee in hoe je een statisch formulier omzet naar een React Hook Form, inclusief meerdere validatoren.

Zie je de video niet verschijnen? Klik dan [hier](#) om de video in een nieuw tabblad te bekijken.

4.5 Conditionele velden

Het kan natuurlijk voorkomen dat we een inputveld alleen willen tonen wanneer de gebruiker specifieke data invoert. Neem een selectbox: wanneer de gebruiker tussen alle opties voor 'anders' of 'overig' kiest, willen we de gebruiker de kans geven om dit te specificeren in een tekstveld. Om dit mogelijk te maken, hebben we een `watch-method` nodig die de waarden van onze selectbox - letterlijk! - in de gaten houdt.

Later we beginnen met een selectBox en het optionele tekst-veld. In deze situatie gaan we er vanuit dat we deze elementen toevoegen aan een bestaand React Hook Form, dus we slaan de initiële opzet daarvan over. Wanneer we beide form-elementen registreren, ziet dat er zo uit:

```
<label htmlFor="referrer">
  Hoe heb je dit recept gevonden?
  <select id="referrer" {...register("found-through")}>
    <option value="google">Google</option>
    <option value="friend">Vriend</option>
    <option value="advertisement">Advertentie</option>
    <option value="other">Anders</option>
  </select>
</label>
<input
  type="text"
  {...register("found-through-anders")}>
</input>
```

Nu kunnen we de watch-methode destructuren uit de bestaande useForm-aanroep:

```
const { handleSubmit, formState: { errors }, register, watch } = useForm();
```

Nu kun je een variabele aannemen waarin het resultaat van de watch-methode wordt opgeslagen. Hier zal altijd de waarde (in ons geval de string in het value-attribuut) van de gekozen optie komen te staan. Super handig, want daarmee kunnen we bepalen wanneer het tekst-veld weergegeven moet worden of niet. Maar het is ook mogelijk om meerdere velden in de gaten te houden:

```
// Optie 1: houd de waarde van één veld bij
const watchSelectedReferrer = watch('found-through');
// Optie 2: houd de waarde van alle meerdere velden bij:
const watchAllFields = watch(['found-through', 'found-through-anders']);
// Optie 3: houd de waarden van alle velden bij:
const watchAllFields = watch();
```

Gedien we slechts één veld in de gaten willen houden, geven wij de naam van het invloedveld mee. Daardoor kunnen we nu met conditionele renderen ook voor zorgen dat het andere invloedveld alleen wordt weergegeven wanneer de geselecteerde waarde op 'anders' wordt gezet.

```
function App() {
  const { handleSubmit, formState: { errors }, register, watch } = useForm();
  const watchSelectedReferrer = watch('found-through');

  return (
    <form onSubmit={handleSubmit(handleFormSubmit)}>
      /* input velden en selectbox */
      {watchSelectedReferrer === "other" &&
        <input
          type="text"
          {...register("found-through-anders")}>
      }
    </form>
  );
}
```

Dit syntax zorgt ervoor dat iedere keer als de gebruiker een andere keuze maakt, deze conditie opnieuw wordt geëvalueerd. Het extra veld verschijnt en verdwijnt dus bij iedere wijziging in de selectie!

4.6 Formulier opties

Onze formulieren komt met een aantal standaardinstellingen. Gelukkig kunnen we die gemakkelijk wijzigen als we willen! Misschien wil je sommige invloedvelden al vullen met waarden uit de backend (prefilling), of de validatie al laten plaatsvinden zodra de gebruiker begint met typen, in plaats van te wachten tot er op "verzenden" gedrukt wordt.

Het wachten met validatie tot de gebruiker het formulier probeert te verlaten, is standaard functionaliteit van React Hook Form. Er zijn echter ook andere mogelijkheden:

- onblur: pas waarden weer op de submit-loop geldig word (standaard);
- onblur: valideren zodra een veld weer heet ("verlaten");
- onChange: valideren toewijt de gebruiker het type is, dus op ieder onChange-event. Let op: het formulier zal bij elke toetsaanslag opnieuw valideren, dus dit heeft impact op de performance van jouw formulier. Bij een klein formulier (zoals alleen gebruikersnaam en wachtwoord) is dit geen probleem, maar bij een groter formulier wordt dit niet aangeraden;
- onTouch: valideren zodra een gebruiker een veld voor de eerste keer heet ("verlaat"), en daarna bij elke toetsaanslag;

In het geval dat we dit willen wijzigen, kunnen we de mode-instelling van ons formulier overschrijven:

```
function App() {
  const { handleSubmit, formState: { errors }, register } = useForm({ mode: 'onBlur' });
}
```

Prefilling

Op dezelfde manier kunnen we ons formulier ook vullen met waarden, door de defaultValues instelling te overschrijven:

```
function App() {
  const { handleSubmit, formState: { errors }, register } = useForm({
    mode: 'onBlur',
    defaultValues: {
      foundThrough: 'advertisment',
      age: 12,
    },
  });
}
```

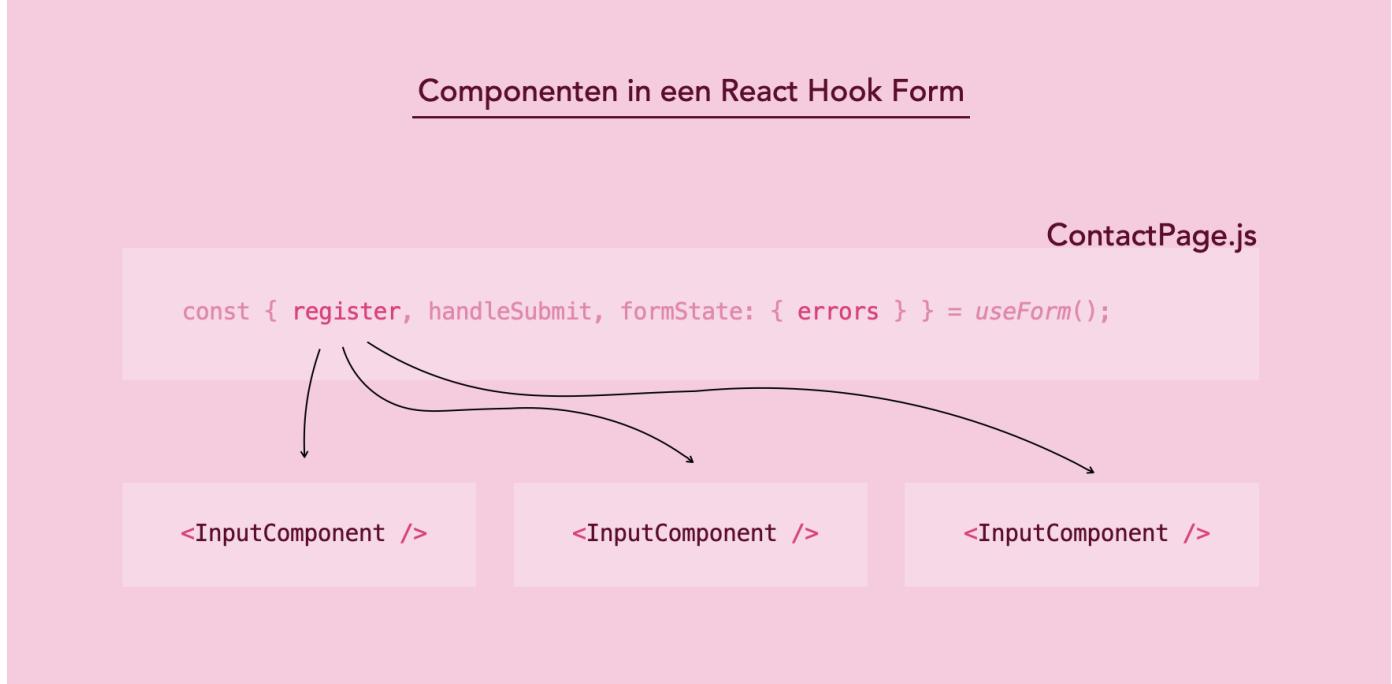
Let op: dat de keys die we daar gebruiken overeen moeten komen met de name-attributen (dus het eerste argument dat we aan elke register-functie meegeven) van onze inputvelden!

Je hebt nu gelijk hoe je de meest gebruikte functionaliteiten van een formulier kunt toepassen. Er zijn natuurlijk nog veel meer mogelijkheden, maar die kun je zelf gaan ontdekken op het moment dat je zo'n edge case tegenkomt. Neus dan eens rustig door de [volledige documentatie](#) van React Hook Form: deze is super overzichtelijk en altijd voorzien van duidelijke code-voorbeelden!

4.7 Van elementen naar componenten

Wanneer je jouw invloedvelden wilt omzetten naar herbruikbare componenten die je kunt gebruiken binnen React Hook Form, hoef je slechts een kleine aanpassing te maken! Hierbij is het belangrijk dat je uitgaat van het volgende principe:

- Het insteken van het formulier (het aanroepen van useForm) doe je slechts één keer per formulier;
- De entiteiten die je daaruit haalt (zoals register, errors, etc.) geef je door aan de input-componenten die je in het formulier gebruikt;
- Je kunt componenten maken van jouw input-elementen, maar niet van de form-tag. Deze staat altijd op de pagina waar je useForm aanroept, zodat je controle hebt over de data wanneer deze wordt verstuurd.



In onderstaand voorbeeld werken we nog in App.js, maar in de toekomst zal je deze code waarschijnlijk op een aparte pagina (zoals ContactPage.js of Register.js) plaatsen. Laten we eens kijken hoe we de invoer- en label-combinatie van onderstaand formulier kunnen opzetten als component. Dit is onze oude situatie:

```
function App() {
  const { register, handleSubmit, formState: { errors } } = useForm();
  function handleFormSubmit(data) {
    console.log(data);
  }
  return (
    <form onSubmit={handleSubmit(handleFormSubmit)}>
      <label htmlFor="name-field">
        Naam:
        <input
          type="text"
          id="name-field"
          {...register("name", {
            required: {
              value: true,
              message: 'Naam is verplicht',
            }
          })}
        />
      </label>
      {errors.name && <p>{errors.name.message}</p>}
      <button type="submit">
        Versturen
      </button>
    </form>
  );
}
export default App;
```

De useForm-methode wordt aangeroepen in App.js. Dit houden we zo, gezien we op die manier alle benodigde methodes kunnen doorgeven. Ook de form-tag en de handleSubmit-methode laten we op dit niveau staan. Voor de HTML van het label-, input- en p-element zullen we één herbruikbaar component maken. Hiervoor maken we eerst een nieuw bestand aan, genaamd InputComponent.js, en plakken daar onze HTML in:

```
// InputComponent.js
function InputComponent(props) {
  return (
    <label htmlFor={props.htmlFor}>
      Naam:
      <input
        type={props.type}
        id={props.id}
        {...props.register}
        required={props.required}
        value={props.value}
        message={props.message}
      />
    </label>
    {props.errors && <p>{props.errors.message}</p>}
  );
}
export default InputComponent;
```

Uiteraard geeft dit nu een hoop foutmeldingen. Dus laten we eens kijken naar de stukjes data die dit component moet ontvangen om weer te kunnen werken:

- Het type van invoerveld
- De naam van het invoerveld
- De label-tekst die boven het invoerveld staat;
- De id van het invoerveld (dit moet overeenkomen met het htmlFor-attribuut op het label);
- Een object met validateregels voor dit invoerveld;
- De register-methode en errors-object van React Hook Form;

We gaan er daarom voor zorgen dat wanneer we ons component gebruiken in App.js, het al deze data aangeleverd krijgt via properties en callback-properties:

```
// App.js
<InputComponent
  inputType="text"
  inputName="name"
  inputId="name-field"
  inputLabel="Naam:"
  validationRules={{
    required: {
      value: true,
      message: 'Naam is verplicht',
    }
  }}>
</InputComponent>
// we geven de register-methode van react-hook-form mee onder de naam register
register={register}
// we geven het errors-object van react-hook-form mee onder de naam errors
errors={errors}
/>
```

Let op: we geven de register-methode dus onder dezelfde naam (register) door. We hadden dit ook als bananen+register kunnen doen, maar op deze manier blijft de schijfje welke waarde we de register-methode gaan toepassen in het component. Daarnaast zie je ook dat we bij het doorgeven van de properties de methodes niet aanroepen en er ook geen argumenten aan meegeven. Dit doen we namelijk pas in het component zelf.

```
function InputComponent({ inputType, inputName, inputLabel, inputId, validationRules, register, errors }) {
  return (
    <label htmlFor={inputId}>
      {inputLabel}
      <input
        type={inputType}
        id={inputId}
        {...register(inputName, validationRules)}
      />
    </label>
  );
}
```

```
</label>
);
```

Omdat we de register-methode onder de naam register hebben meegegeven, kunnen we hem net zo omschrijven als wanneer je geen componenten zou gebruiken. De methode wordt nog steeds op dezelfde manier toegepast en krijgt nog steeds twee argumenten mee: de naam van het veld en het object met validatorenregels. Om de errors goed weer te geven, zullen we gebruik moeten maken van de `blockstaak-notatie` in plaats van de `pure-notatie`. We willen immers een variabele als object-key gebruiken:

```
{errors[inputName] && <p>{errors[inputName].message}</p>}
```

En dat ziet er dan ten slotte zo uit:

```
// InputComponent.js
function InputComponent({ inputType, inputName, inputLabel, inputId, validationRules, register, errors }) {
  return (
    <>
      <label htmlFor={inputId}>
        {inputLabel}
        <input
          type={inputType}
          id={inputId}
          {...register(inputName, validationRules)}
        />
      </label>
      {errors[inputName] && <p>{errors[inputName].message}</p>}
    </>
  );
}

export default InputComponent;
```

```
// App.js
import InputComponent from './components/InputComponent';

function App() {
  const { register, handleSubmit, formState: { errors } } = useForm();
  function handleFormSubmit(data) {
    console.log(data);
  }

  return (
    <form onSubmit={handleSubmit(handleFormSubmit)}>
      <InputComponent
        inputType="text"
        inputName="name"
        inputId="name-field"
        inputLabel="Naam"
        validationRules={{
          required: {
            value: true,
            message: 'Naam is verplicht',
          }
        }}
        register={register}
        errors={errors}
      />
      <button type="submit">
        Versturen
      </button>
    </form>
  );
}
```

Op deze manier kun je oneindig veel Input-componenten blijven toevoegen die je dezelfde register-methode en errors-object doorgeeft, omdat ze onderdeel zijn van hetzelfde formulier.

5. Routing

5.1 Inleiding

In dit hoofdstuk ga je alles leren over het bespelen van routing in React. En hiermee bedoelen we alles dat nodig is om jouw gebruikers de juiste content te laten zien op de juiste pagina. Bij traditionele applicaties hadden we hier weinig voor te doen, deze werden namelijk gerenderd aan de server kant: **Server Side Rendering**. React applicaties worden - net als andere frameworks zoals Vue en Angular - gerenderd aan de kant van de gebruiker: **Client Side Rendering**. Maar wat betekent dat eigenlijk?

Vroeger bestond Client Side Rendering helemaal niet! Als je www.out.nl in de adresbalk typeert, werd er een verzoek naar de server om de homepage weer te geven. De server ging dan op zoek naar de specifieke pagina, gaf alles in één kant-en-klar HTML-bestand en stuurde dat terug naar de browser. Navigeerde de gebruiker naar de contactpagina? Dan ging de server opnieuw aan de slag om alle HTML-, CSS- en JavaScript voor de contactpagina bij elkaar te zoeken en die vervolgens kant-en-klar aan de browser te serveren. Iedere keer dat de gebruiker op een link klikte of een actie uitvoerde (zoals het versturen van een contact-formulier) werd er een nieuwe pagina opgeroepen en in de hele applicatie verwerkt. En omdat het versturen van HTML veel langzamer is dan losse statische data, kon dit soms traag gaan.

Met de komst van Frontend Frameworks werd Client Side Rendering geïntroduceerd: dit houdt in dat de volledige applicatie wordt opgeboord aan de kant van de gebruiker. Stellem dat je al lang opent! Want wanneer je een React applicatie opstart, wordt de lege `index.html` vervangen met content, door alle JavaScript en CSS te bundelen en in de lege `<div id="root"></div>` te injecteren. Wanneer je www.out.nl in de adresbalk invoert, wordt er bij een Client Side applicatie een verzoek naar de server gedaan om alle losse HTML, CSS en JavaScript op te vragen. Vervolgens wordt de applicatie eenmalig opgeboord in de browser: alle pagina's in één keer. Om deze reden noemen we React applicaties **Single Page Applications** (ook wel SPAs genoemd), omdat je wil len doen bij je collega's).

Hmmm... Maar we willen de gebruiker toch niet alle pagina's tegelijk laten zien?

Dat klopt! Daarom zorgen we er met React Router voor dat we alleen de pagina weergeven die correspondeert met de url in de adresbalk. Conditioneel pagina's renderen, zeg maar. Maar wanneer je op deze manier tussen pagina's wisselt, kom je voor een aantal uitdagingen te staan. Die uitdaging ligt 'niet per se in het laden van de content, maar in het bieden van een gebruikerservaring die overeenkomt wat gebruikers gewend zijn'. Want wanneer zij onze applicatie gebruiken, verwachten ze dat:

- De URL van de pagina verandert met helpende wat er op dat moment beschikt;
- De browser-inhaken (vorige en volgende pagina's) behouden;
- Ze rechtsklikken naar een specifieke weergave (ook wel bekend als `deep linking`) kunnen navigeren met de juiste URL.

Wanneer we de frontend bouwen van een server-side applicatie, krijg je deze functionaliteit gratis bij. Maar omdat een Single Page Application technisch gezien nooit daadwerkelijk van pagina of url verandert, moeten we een aantal stappen ondernemen om dit gedrag na te bootsen. Als je nu al zweverige handpalmen hebt gekregen - geen zorgen! Hier hebben we React Router voor.

In de volgende paragrafen leer je hoe je React Router kunt toepassen om routing te implementeren, gebruikers naar andere pagina's te sturen en hoe je beveiligde-, generieke- en dynamic-routes bouwt.

5.2 Routing structuur implementeren

Nu we onze applicatie gaan uitbreiden met meerdere pagina's, is het belangrijk om eerst naar de indeling van ons project te kijken. Het is conventie om al onze componenten ieder netjes aan hun eigen stylesheet te koppelen en deze samen in een mapje te stoppen:

```
src
└── components
  └── button
    └── Button.js
  └── header
    └── Header.js
```

Wanneer we pagina's gaan maken, zijn dit in feite ook gewone componenten die ieder hun eigen stylesheet gebruiken. Om duidelijk aan te geven dat we het over pagina's hebben, stoppen we ze ook samen in een mapje:

```
src
└── pages
  └── home
    └── Home.js
    └── Home.css
  └── faq
    └── Faq.js
    └── Faq.css
```

Een pagina, zoals `Home.js`, bevat een mix van normale HTML-elementen en jouw eigen componenten en maakt hier een samenhangend geheel van. Deze pagina geven we vervolgens weer door deze te importeren in `App.js`:

```
import React from 'react';
import Home from './pages/home/Home';

function App() {
  return (
    <>
      <Home/>
      /* Eventuele andere pagina's ... */
    </>
  );
}
```

Onderstaande video neemt je mee in de implementatie van React Router in een bestaand project. Zie je de video niet verschijnen? Klik dan [hier](#) om de video in een nieuw tabblad te bekijken.

React wordt dus niet automatisch geleverd met routing. In theorie zou je dit zelf kunnen maken, maar het is veel handiger om de [React Router packages](#) te gebruiken. Wanneer je dit wil gaan toepassen in jouw project, zul je het altijd eerst even moeten installeren door het volgende commando in te voeren in de terminal:

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

Daarna moeten we altijd het `<BrowserRouter>`-component om onze volledige applicatie wikkelen. Op deze manier kunnen alle elementen gebruik maken van de functionaliteiten die React Router biedt.

Let op: het is conventie om `BrowserRouter` te hernoemen naar `Router`, maar het is niet verplicht. We wikkelen dit component om onze applicatie in `index.js`:

```
// index.js
import { BrowserRouter as Router } from 'react-router-dom';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <Router>
```

```
<App/>
</Router>
</React.StrictMode>
);
```

Routes definieren

Vervolgens kun je gaan definieren welke URL's de gebruikers mogen bezoeken en, belangrijker nog, welke pagina's zij dan te zien krijgen. Hiervoor gebruik je één `<Route>`-component, waarin je alle afzonderlijke `<Route>`-componenten definiert. Elke `<Route>`-component verwacht twee properties:

1. Het path, de wijn de url. Hierin hoef je niet de basis-url (zoals `http://localhost:3000` of `www.your-website.nl`) mee te nemen: dit doet React automatisch voor ons. Je gebruikt dus alleen de relatieve url die daaraan vastgeplakt wordt, zoals `/ voor home`, `/contact voor de contactpagina en` * voor alles dat niet overeenkomt met de bovenstaande routes. Uiteraard mag je iedere url kiezen die je wilt. De FAQ-pagina op `http://localhost:3000/faaanaar?`
2. Het element, waaraan je de pagina meegeeft die mag worden weergegeven.

```
// App.js
import { Routes, Route } from 'react-router-dom';
function App() {
  return (
    <> /* Toekomstige menu balk... */
    <Routes>
      <Route path="/" element={<Home/>}/>
      <Route path="/faq" element={<FAQ/>}/>
      <Route path="/testimonials" element={<Testimonials/>}/>
      <Route path="/" element={<NotFound/>}/>
    </Routes>
    /* Eventuele footer ... */
  );
}

In dit geval mag alleen de toekomstige menu balk buiten het <Routes>-component blijven staan: deze staat immers boven iedere pagina en hoeft niet te wisselen. Dit was ook het geval geweest indien we een pagina-footer zouden gebruiken. Let op: heb jij jouw interface zo ontworpen dat sommige pagina's één navigatie of footer hebben en sommige wel? Of ziet de footer er op sommige pagina's anders uit? In dat geval zou je het navigatie- en footer element op iedere pagina afzonderlijk moeten invoegen, in plaats van eenmalig in App.js.
```

Links

Fantastisch, we hebben onze routes gedefinieerd, maar dat kan de gebruiker natuurlijk niet ruiken. We willen de gebruiker daarom met links kunnen doorsturen naar de juiste pagina's. Wanneer we de gebruiker binnen onze applicatie naar een andere pagina sturen, doen we dit niet meer met de gebruikelijke anchor-tags (`<a>`), maar met het speciale `Link`-component van React Router.

HTML	React
<code>hier</code>	<code><Link to="/">hier</Link></code>

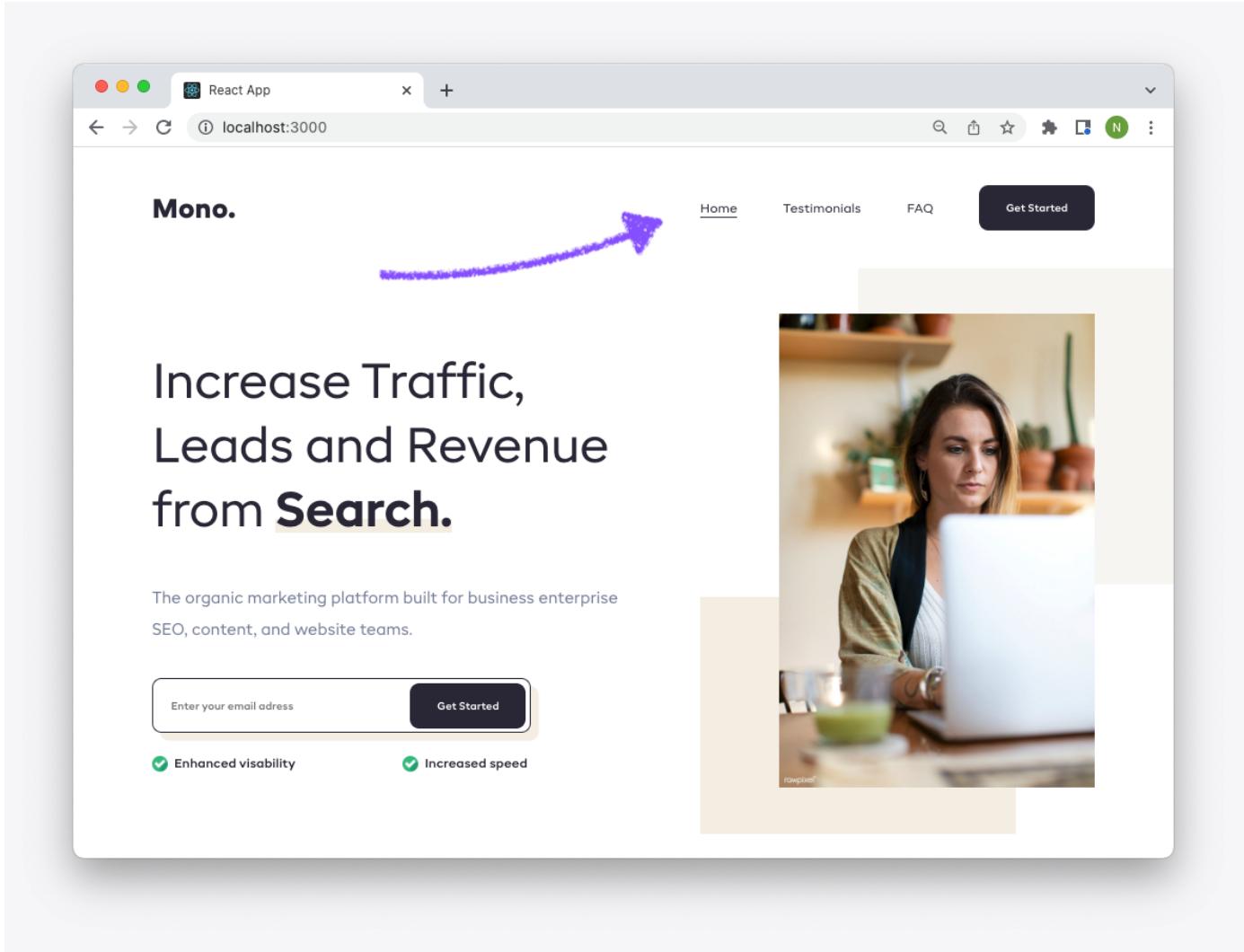
```
// Contact.js
import React from 'react';
import { Link } from 'react-router-dom';

function Contact() {
  return (
    <div>
      <h1>Contact pagina</h1>
      <Link to="/about">Naar de "over ons" pagina</Link>
    </div>
  );
}
```

Dit betekent niet dat je nu niet meer een anchor-tag mag gebruiken. De a-elementen zullen we alleen nog gebruiken als we naar webpagina's buiten onze applicatie linken, zoals `www.novi.nl` of `www.google.com`.

NavLink

In theorie kunnen we voor onze navigatie-balk ook `<Link>`-componenten gebruiken. Maar in veel gevallen willen we links in de navigatiebalk een ander uiterlijk geven wanneer de gebruiker op die specifieke pagina zit. Denk bijvoorbeeld aan een donkere markering of een streep.



Om deze reden heeft Router ook `<NavLink>`-componenten: deze zijn zich bewust van de huidige url en weten daarom waarneer ze actief zijn. Om straks te kunnen wisselen, kunnen we alvast twee classes in de CSS klaarzetten: een voor het uiterlijk van een inactive link en een voor het uiterlijk van een active link:

```
.active-menu-link {
  border-bottom: none;
}
.default-menu-link {
  border-bottom: 1px solid black;
}
```

Vervolgens kunnen we onze navigatie voorzien van `<NavLink>`-componenten:

```
// Navigation.js
import React from 'react';
import { NavLink } from 'react-router-dom';

function Navigation() {
  return (
    <nav>
      <ul>
        <li>
          <NavLink to="/">Home</NavLink>
        </li>
        <li>
          <NavLink to="/testimonials">Testimonials</NavLink>
        </li>
        <li>
          <NavLink to="/faq">FAQ</NavLink>
        </li>
      </ul>
    </nav>
  );
}
```

Ook moment gedragen onze `<NavLink>`-componenten zich als normale `<Link>`-componenten. Om de conditionele styling toe te passen, mogen we een anonieme functie aan de `className`-property meegeven (een callback). We moeten eerlijk bekennen dat de manier waarop de makers van React Router dit hebben opgezet, niet bijzonder leesbaar is. Daarom vind je aan het eind van deze paragraaf een uitgebreide uitleg van deze schrijfwijze*, maar het is niet noodzakelijk om dit te begrijpen. Je mag deze syntax ook gewoon "sinnenvast" zoals 't lezen.

```
<NavLink
  className={({ isActive }) => isActive ? 'active-menu-link' : 'default-menu-link'}
  to="/"
  Home
</NavLink>
```

Let op: deze functionaliteit is bedacht door React Router. Dit is geen algemene React toepassing en kun je dus ook niet toepassen op andere componenten.

*Hoe werkt de callback van `NavLink`?

Voor we ingaan op hoe React Router dient dat de conditionele styling moet worden toegepast, gaan we eerst eens kijken op hoe we dat op een normaal element zouden doen. Stel dat we een variabele hebben, genaamd `isActive`, waarin we hebben opgeslagen of de huidige URL overeenkomt met de url waar onderstaand anchor-element naartoe linkt. We zouden het wisselen tussen onze CSS class "active-menu-link" en "default-menu-link" dan ophlossen met een ternary operator:

```
// Navigation.js
function Navigation() {
  const isActive = true;

  return (
    <link className={isActive ? 'active-menu-link' : 'default-menu-link'} to="/">Home</Link>
  );
}
```

In bovenstaand voorbeeld is het achter zo dat we zelf verantwoordelijk zijn voor het bepalen of de link actief is (`const isActive = true;`) of niet (`const isActive = false;`). Om dit te voorkomen, wil React Router dat je een anonieme functie meegeeft aan de `className`-property (een callback). Deze anonieme functie krijgt dan automatisch een object toegevoegd waar deze informatie op staat - een beetje zoals het event-object waarneer je evenlisteners gebruik. Op dat object vind je altijd twee keys:

```
navObject = {
```

```

    isActive: true,
    isPending: false,
}

Dat betekent dat als we de isActive key willen gebruiken, we die via de punt-notatie kunnen aanspreken (navObject.isActive), of direct kunnen destructuren ({isActive});

```

<NavLink className={({ navObject }) => navObject.isActive ? 'active-menu-link' : 'default-menu-link'}>
 // of
 <NavLink className={({ isActive }) => isActive ? 'active-menu-link' : 'default-menu-link'}>

En daarom ziet iedere NavLink er zo uit:

```

<NavLink
  className={({ isActive }) => isActive ? 'active-menu-link' : 'default-menu-link'}
  to="/"
  Home
/>

```

5.3 Doorlinken

In veel gevallen kunnen we de gebruiker op een <Link>-of <NavLink>-component laten klikken om naar een andere pagina te navigeren. Maar wanneer de gebruiker op een button-element klikt, is dat een ander verhaal. We kunnen geen <Link>-component in het onClick event zetten:

```

<button
  type="button"
  onClick={() => console.log("We willen naar /contact! Maar hoe?")}
>
  Ga verder
</button>

```

En kijk ook eens naar het volgende voorbeeld. Hoe moeten we het oplossen wanneer we de gebruiker door willen sturen nadat een formulier ingevuld en verstuurd is?

```

function Login() {
  function onFormSubmit(e) {
    e.preventDefault();
    // ... verstuur de data naar de backend ...
    // ... en als dat gelukt is, willen we door naar "/profile"!
  }

  return (
    <form onSubmit={onFormSubmit}>
      {/* invoervelden en submitbutton... */ }
    </form>
  );
}

```

Je merkt het al, we hebben ook een handmatige manier nodig om gebruikers te kunnen doorsturen zonder dat zij expliciet op een hyperlink klikken. In dit soort gevallen gebruiken we de useNavigate-methode van React Router. Wellicht komt deze naamgeving je al wat bekend voor: het feit dat er "use" voor staat, betekent dat het een hook is (net als useState, maar dan gebouwd door React Router).

Gebruikers doorsturen zonder hyperlink

1. Importeer de useNavigate-hook

```
import { useNavigate } from "react-router-dom";
```

2. Roep de hook eenmalig aan aan bovenin het component

```
const navigate = useNavigate();
```

3. Gebruik de navigate-methode zo vaak als je wil om gebruikers door te sturen naar een nieuw pad

```
navigate("/profile");
```

Net als bij useState, roepen we de hook eenmalig aan (bovenin ons component of pagina). Wat we hieruit terugkrijgen is één methode, navigate, die we vervolgens op ieder gewenst moment kunnen aanroepen door er het nieuwe, gewenste pad aan mee te geven:

```

import { useNavigate } from "react-router-dom";
function Login() {
  const navigate = useNavigate(); // We roepen de hook eenmalig aan;

  function onFormSubmit(e) {
    e.preventDefault();
    navigate("/profile"); // <-- We kunnen navigate-methode oneindig vaak gebruiken!
  }

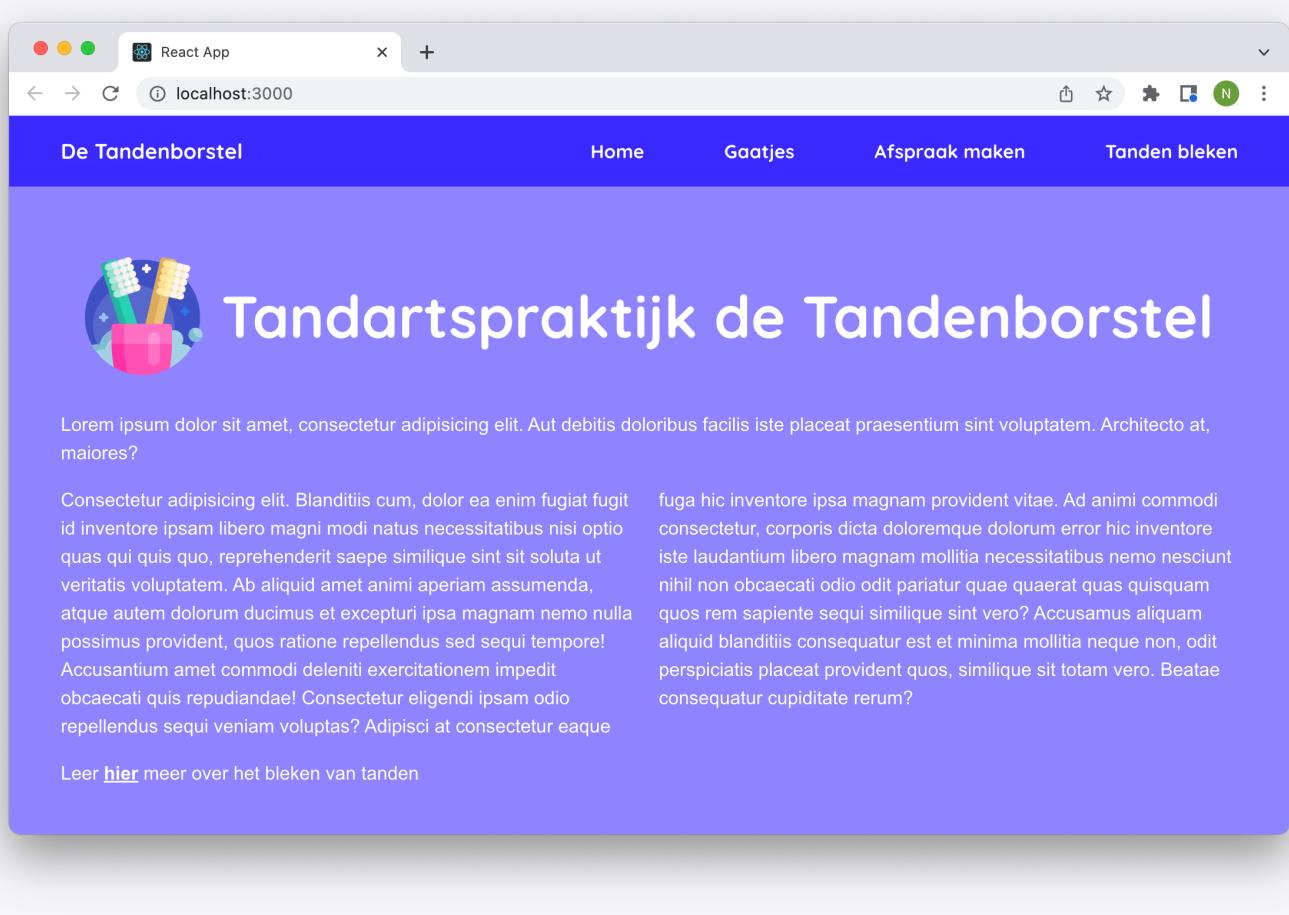
  return (
    /* formulier... */
  );
}

```

<Link> is dus een component dat we gebruiken als hyperlink en useNavigate is een hook die we gebruiken binnen onze logica om gebruikers door te sturen naar een ander pagina.

5.4 Opdracht: tandartspraktijk

Je gaat zelf routing implementeren voor de website van Tandartspraktijk de Tandenborstel. Dit ga je doen door gebruik te maken van de package React Router. Om de opdracht te maken kun je de opdracht clonen of downloaden naar jouw eigen computer via [deze GitHub repository](#). De uiterwerkingen staan op de branch uitwerkingen.



Applicatie starten

Als je het project gedownload hebt naar jouw lokale machine, installeer je eerst de node_modules door het volgende commando in de terminal te runnen:

`npm install`

Wanneer dit klaar is, kun je de applicatie starten met behulp van:

`npm start`

... of gebruik de WebStorm knop (npm start). Open <http://localhost:3000> om de pagina in de browser te bekijken. Begin met het maken van wijzigingen in `src/App.js`; elke keer als je een bestand opslaat, zullen de wijzigingen te zien zijn op de webpagina.

Opdrachtbeschrijving

Onderstaande stappen beschrijven op globale wijze wat er moet gebeuren. Schroom niet om in de vorige paragrafen te speken naar de details of specifieke schrijfwijze:

- Installeer React Router Dom in het project;
- Wilkkel het buitenste React Router element om de applicatie in `Index.js`;
- Implementeer routing voor alle vier van de pagina's in `App.js`:

 - a. De homepage;
 - b. De afsprakenpagina op `/afspraken`;
 - c. De caudlepagina op `/gaatjes`;
 - d. De websitepagina op `/tandenbleken`.

- Check de navigatie en de menu-items in de sidebarbalk of de url aan te passen. Veranderen de pagina's?
- Op de homepage en afsprakenpagina staat onderaan een paraagraaf met het stijltype: 'Leer hier meer over [onderwerp]'. Zorg ervoor dat het woord 'hier' klikbaar wordt en de gebruiker doortruist naar de juiste pagina.
- Zorg ervoor dat de menu-items in `Navigation.js` klikbaar worden en de gebruiker doortruist naar de juiste pagina.
- Zorg ervoor dat het actieve menu-item wit onderstrept wordt, door twee CSS classes te maken in het bijbehorende `Navigation.css` document en deze op de juiste manier te geven aan het `NavLink`-component van React Router.
- Maak een sidebar die de verschillende onderdelen bevat, door de volgende acties te ondernemen:
 - a. Maak een nieuwe component `SideBar.js`;
 - b. Zorg ervoor dat deze functie wordt aangeroepen wanneer er op de button geklik wordt;
 - c. Zorg ervoor dat wanneer deze functie wordt aangeroepen, er "We gaan direct naar de afspraken pagina!" wordt gelogd in de console;
 - d. Stuur de gebruiker daarna door naar de afsprakenpagina;

5.5 Dynamische routes

In iedere programmeertal we zoeken mogelijk volgens het DRY principe: *Don't Repeat Yourself*. Hoe meer code je hergebruikt, hoe beter deze code later weer te onderhouden en aan te passen is.

Stel dat we de website van Albert Heijn zouden moeten bouwen. Ieder product dat zij verkopen heeft een productpagina. Te vinden op een unieke url, die het artikelnummer bevat. Iedere pagina bevat algemene informatie over het product, een afbeelding, de prijs en de ingrediëntenlijst. Albert Heijn heeft meer dan 25 500 producten in het assortiment. Hoeveel pagina's denk je dat we in React nodig hebben om al de producten weer te geven?

Precies maar één.

Net zoals we de componenten hergebruiken, zullen we dat ook doen met gelijnnamige pagina's. Iedere keer als een gebruiker de pagina over sperezienbonen (artikelnummer 395946) bezoekt, wordt de template-pagina gevuld met data over sperezienbonen. Bezoek de gebruiker daarna de productpagina voor griesmeelpudding (artikelnummer 385831), dan wordt dezelfde pagina opnieuw op dezelfde plekken gevuld met informatie over griesmeelpudding. Alleen, hoe weet de pagina nu welk product de gebruiker op dat moment bekijkt?

We halen deze informatie uit de url. We kunnen onze routing-structuur zo opzetten dat we rekening houden met een dynamisch element in de url. Laten we de locaties voor de griesmeel- en sperezienbonen-pagina's naast elkaar leggen:

```
* www.anrproduct/395946 (sperezienbonen)
* www.anrproduct/385831 (griesmeelpudding)
```

We weten dat we op de url /product de productpagina willen laten zien. Maar het laatste deel van de url dat ons vertellen welke productdata er moet worden weergegeven. Dit principe noemen we dynamic parameters. Bij het opzetten van de routing structuur, vertellen we React Router rekening te houden met een dynamische parameter door de : te gebruiken:

```
// App.js
<Routes>
  <Route path="/product/:id" element={<ProductPage/>} />
</Routes>
```

We geven onze dynamische parameter ook altijd een naam, zodat we de waarde die eraan komt te staan later kunnen aanspreken. Hoe je de dynamische parameter noemt, maakt niets uit! Zolang er maar een dubbele punt voor staat, zou je ook path="/product/:articlnummer" of path="/product/:bladjebla" mogen gebruiken.

Dit zorgt ervoor dat de gebruiker nu zowel /product/395946 als /product/395946 als /product/test zou kunnen bezoeken. Echter, wanneer de productpagina geladen wordt, moet deze pagina wel weten welke data er opgevraagd moet worden. We moeten deze dynamische waarde dus weer kunnen opvragen uit de url door middel van de useParams-hook.

```
// ProductPage.js
import React from "react";
import { useParams } from "react-router-dom";

function ProductPage() {
  const { id } = useParams();

  return (
    <div>Het productnummer is {id}</div>
  )
}
```

Op basis van het artikelnummer, kunnen we nu de juiste data ophalen. Hoe het fetchen van data werkt in React, leer je later in deze cursus.

5.6 Beveiligde routes

Wanneer we een url beveiligen, betekent dit dat we dat we altijd eerst controleren of er wel voldaan wordt aan de conditie voordat we de pagina weergeven. Een conditie kan bijvoorbeeld zijn dat de gebruiker ingelogd moet zijn, of dat de gebruiker admin-rechten moet hebben. Wanneer dit niet zo is, mag het niet zo zijn dat de gebruiker naar een willekeurig scherm zal kijken. We willen de gebruiker dan netjes terugsturen naar de home- of intropagina.



Nu is het zo dat de `element`-property van het `<Route>`-component alleen elementen of componenten accepteert, zoals `<FAQ>` of `<p></p>`. Daarom kunnen we de `useNavigate`-hook hier niet gebruiken. Een `<Link>`-component is ook geen optie, want we willen immers dat de gebruiker automatisch wordt doorgestuurd. Daarom heeft React Router voor deze toepassing een `<Navigate>`-component gemaakt die we kunnen gebruiken:

```

// App.js
import { Routes, Route, Navigate } from 'react-router-dom';
function App() {
  const isLoggedIn = false;
  return (
    <Routes>
      <Route path="/" element={<Home/>}>
        <Route path="/faq" element={isLoggedIn === true ? <FAQ/> : <Navigate to="/" />} />
      </Routes>
  );
}
  
```

Wanneer er aan de conditie wordt voldaan wordt de `<FAQ>`-pagina weergegeven. Zo niet, dan wordt het `<Navigate>`-component weergegeven die de gebruiker in een nanoseconde doorstuurt naar een andere route. De gebruiker zal dit component dus nooit zien.

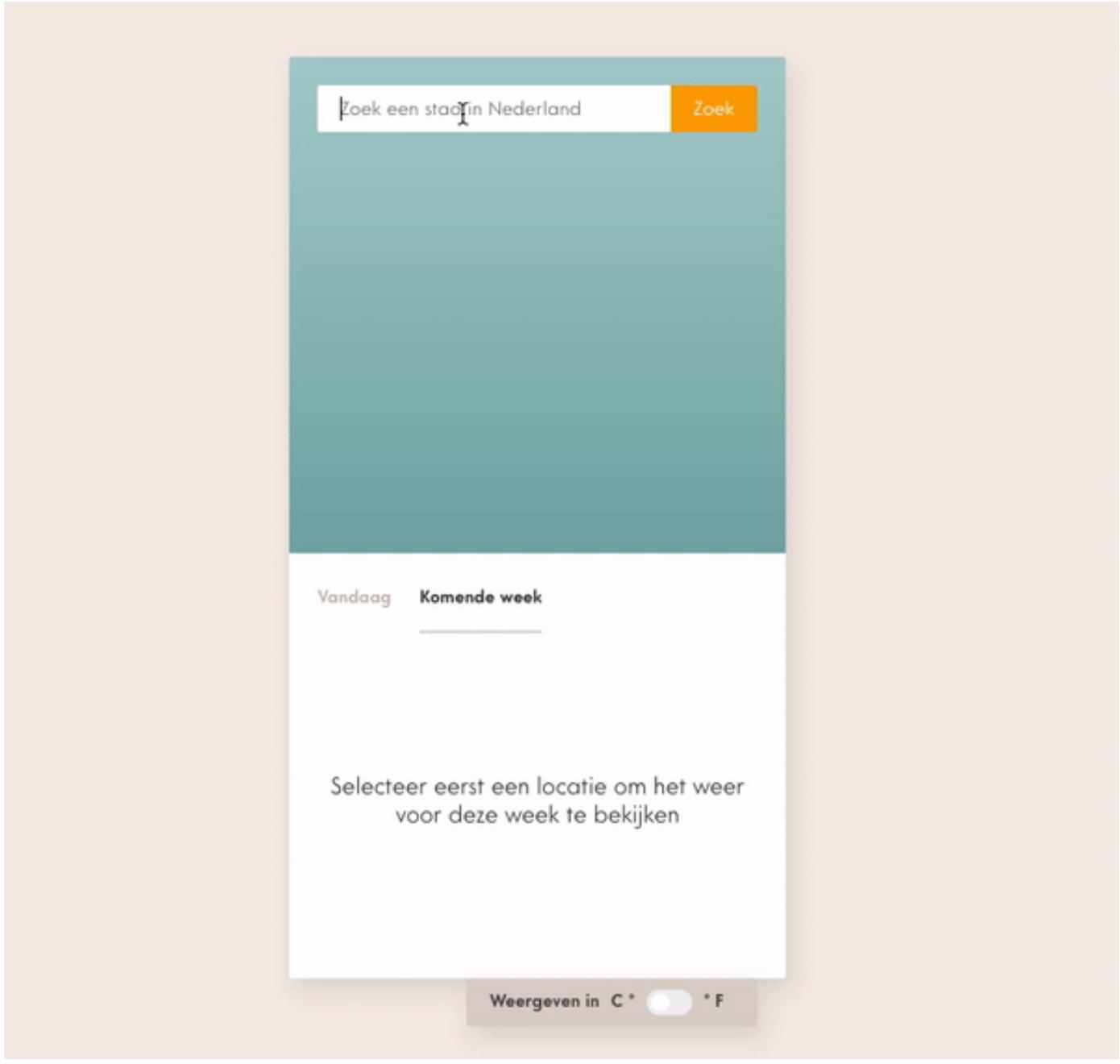
Doordenkervraag: In theorie zouden we ook gewoon de `<Home />`-pagina kunnen weergeven als er niet aan de conditie voldaan wordt. Waarom denk jij dat onderstaand stukje code geen correcte implementatie is?

```
<Route path="/faq" element={isLoggedIn === true ? <FAQ/> : <Home/>}/>
```

6. API's en data requests

6.1 Inleiding

In dit hoofdstuk gaan we leren hoe we data ophalen van een externe bron en dit op de juiste manier kunnen gebruiken in onze frontend. We gaan eerst bekijken hoe verschillende typen requests eruit zien en daarna gaan we een applicatie bouwen die het weer voorstelt en zijn data verkrijgt via een externe API. In deze applicatie kan een gebruiker naar een stad in Nederland zoeken en op die manier de voorprognose voor vandaag en de rest van de week bekijken.



Het volgen van de uitleg en tutorial werkt het best als je meedoeft in de boilerplate die we voor je hebben opgezet. Je kunt de GitHub repository hier vinden:

[Weather app tutorial](#)

Clone de GitHub repository naar jouw computer en navigeer naar de projectmap via de terminal of command prompt, of via de ingebouwde terminal van jouw editor. Vergat niet eerst een `npm install` te draaien zodat alle benodigde node modules worden geïnstalleerd. Daarna kun je de applicatie starten met het commando:

`npm start`

In de boilerplate zie je dat we al een aantal componenten voor je hebben opgezet, zoals de zoekbalk (`<SearchBar />`), de navigatie (`<TabBarMenu />`), een weer-block (`<WeatherDetail />`) en de slider om te wisselen tussen Celsius en Fahrenheit (`<MetricSlider />`). Deze vind je terug in de map genaamd `components`.

Omdat deze applicatie een hoop CSS bevat, hebben we elk component in een aparte map gestopt met het bijbehorende CSS bestand erbij. Zo voorkomen we dat we één gigantische lap CSS in `App.css` zetten en houden we de styling overzichtelijk en modular.

6.2 Axios request types

Voordat we in onze weer-applicatie duiken, is het belangrijk om de syntaxis van asynchrone functies en axios nog eens goed te bestuderen. We gebruiken asynchrone functies om data op te halen, omdat het wel een seconde kan duren voor de database de juiste informatie voor je gevonden heeft. We weten immiddels ook dat requests op verschillende manieren fout kunnen gaan, waardoor we altijd een `try/catch`-blok nodig hebben. In het `try`-blok maken we het request, en wanneer dit mis gaat, zal de code automatisch "overstappen" naar het `catch`-blok.

```
async function fetchData() {
  try {
    let result = await <insert-request-here>;
  } catch(e) {
    console.error(e);
  }
}
```

Een request bevat altijd een `endpoint`, dit is het adres waar mee we de data van- of naar de server sturen. In het geval dat we extra gegevens willen meeturen, stoppen we deze data in de `request body`. Als we daarnaast nog extra specificaties aan ons request willen toevoegen, doen we dit in de `HTTP Header` van het request (ook wel `request header` genoemd). Wanneer en hoe je deze concepten toepast, leer je in de volgende paragrafen.

GET-requests

Een GET-request is altijd bedoeld om data op te halen, zoals bijvoorbeeld zoekresultaten, een token, of een profielfoto. Je kunt een axios GET-request maximaal twee argumenten meegeven, namelijk:

1. Het endpoint waarvanwe de data willen ophalen (`url`)
2. Een configuratie object met eventuele extra specificaties (`options`)

```
axios.get(url [, config])
```

Dit configuratie-object is de `request header`. Dit object bevat extra data over het request dat we maken. Vaak is dit echter niet nodig! Een simpel GET-request zonder specificaties ziet er zo uit:

```
const result = await axios.get('https://link-naar-endpoint.nl');
```

Wanneer we wel een request header willen meesturen, omdat we bijvoorbeeld willen specificeren in welk formaat de input- en output data verwacht wordt, voegen we dit object gewoon toe als tweede argument:

```
const result = await axios.get('https://link-naar-endpoint.nl', {
  'Accept': 'application/json',
});
```

Hiermee zeggen we: "We verwachten dat de response terugkomt als JSON". Mocht het endpoint beveiligd zijn, is dit ook de plek waar je bijvoorbeeld een autorisatie-token meesturen.

POST-requests

Een POST-request is altijd bedoeld om data te versturen, zoals bij het registreren van een nieuwe gebruiker, het plaatsen van een review, of het uploaden van een foto. In tegenstelling tot een GET-request, sturen we bij een POST-request dus altijd data mee. Je kunt een axios POST-request maximaal drie argumenten meegeven, namelijk:

1. Het endpoint waarnaar we de data willen versturen (verplicht)
2. Het data-object met alle gegevens die we willen verzenden (verplicht)
3. Een configuratie object met eventuele extra specificaties (optioneel)

```
axios.post(url, data [, config])
```

Het data object is de *request body*. In het geval dat een nieuwe gebruiker gegevens heeft ingevuld om een nieuw account aan te maken, zouden we dit op de volgende manier versturen:

```
const result = await axios.post('https://link-naar-endpoint.nl', {
  username: 'piet1980',
  email: 'piet_pietersen@gmail.com',
  password: '123456',
});
```

Wanneer we een request header aan een POST-request willen toevoegen, zal dit het tweede object zijn:

```
const result = await axios.post('https://link-naar-endpoint.nl', {
  username: 'piet1980',
  email: 'piet_pietersen@gmail.com',
  password: '123456',
}, {'Content-Type': 'application/json'});


```

Hiermee zeggen we: "De data die we meesturen is geformat als JSON". Mocht het endpoint beveiligd zijn, is dit ook de plek waar je bijvoorbeeld een autorisatie-token meesturen.

PUT- en DELETE-requests

Een PUT-request is bedoeld om bestaande data te wijzigen. Omdat je altijd nieuwe data meesturen om bestaande data mee te overschrijven, heeft een PUT-request dezelfde syntax als een POST-request. Het endpoint en request body zijn verplicht, de request header is optioneel:

```
axios.put(url, data [, config])
```

Een DELETE-request is bedoeld om bestaande data te verwijderen. Omdat je hiervoor geen data meesturen (het endpoint wijst al naar de data die verwijderd moet worden) heeft een DELETE-request dezelfde syntax als een GET-request. Het endpoint is verplicht, de request header is optioneel:

```
axios.delete(url [, config])
```

6.3 API keys

Nu we meer te weten zijn gekomen over de verschillende requests die we met axios kunnen maken, kunnen we verder met onze weer-applicatie. We hoeven gelukkig niet zelf te verzinnen wat voor weer het morgen in Utrecht wordt. Hiervoor kunnen we gebruik maken van de gegevens van OpenWeather; zij stellen deze informatie beschikbaar via hun API. In de cursus JavaScript heb je geleerd dat APIs interactie mogelijk maken tussen verschillende applicaties door de overschrijving van gegevens van systeem naar systeem te faciliteren. Een API is de "über" die zorgt dat jouw bestelling bij de keuken komt en jouw gerecht weer eet.

Hoe we deze "bestelling" maken, verschilt per API. OpenWeather heeft vastgesteld dat we op de volgende manier gegevens in de database mogen oproepen als we het huidige weer voor een locatie willen ontvangen:

```
GET https://api.openweathermap.org/data/2.5/weather?q=(NAAM STAD),nl&appid=(JOUW API KEY)&lang=en
```

Zoals je ziet, zit er een hoop informatie in dit endpoint verscholen. We specificeren de stad waarover we de gegevens willen ontvangen en laten ook weten wie we zijn, door middel van de API key. De database van OpenWeather gaat onze data dan verzamelen (zoals de klok ons elke dag bereidt) en stuurt ons een groot JSON object terug (onze pasta bolognese).

API keys worden gebruikt ter identificatie

API-keys worden gebruikt om bij te houden hoe de API wordt gebruikt, maar ook om kwaadwillig gebruik of misbruik van de API's te voorkomen. De API-key fungeert vaak als een unieke identificatiemethode én als een geheime token voor authenticatie. Zie het als het tafelnummer waardoor de ober weet voor wie de bestelling is. Op deze manier is het ook mogelijk om specifieke toegang tot de API te verlenen op basis van de identiteit van de gebruiker. Wij gaan bijvoorbeeld alleen gebruik maken van de gratis onderdelen van OpenWeather. Op basis van onze API-key kan OpenWeather zorgen dat wij geen requests kunnen maken naar betaalde endpoints.

Om de tutorial te maken, heb je een API-key nodig. Deze kun je gemakkelijk verkrijgen door jezelf aan te melden bij OpenWeather. Doe dat via de volgende link (we zullen gebruik maken van het gratis "plan"). Als je hierna naar de tab "API keys" gaat, kun je hier jouw persoonlijke API key vinden. Denk hier je straks nodig, dus hou hem even bij de hand.

You can generate as many API keys as needed for your subscription. We accumulate the total load from all of them.

Key

Name

Create key

d85e6256cb39361faf2b9dacc4eb33f4

Default



API key name

Generate

In App.js vind je de header met wat dummy tekst erin ("Bewolkt, 14 graden") en een nogal ongemakkelijke button die we alleen gaan gebruiken om de eerste stappen makkelijker te maken. In dit bestand gaan we beginnen!

Makkelijk bovenaan in het bestand App.js een variabele apiKey aan en stop jouw API key daarin. Zo kunnen we hem straks makkelijk ook op andere plekken gebruiken:

```
const apiKey = 'd85e6256cb39361faf2b9dacc4eb33f4';
function App() {
  // ...
}
```

Let op: als je de API key uit dit voorbeeld probeert te gebruiken zal de applicatie niet werken. Zorg dus dat je een eigen, unieke API key gebruikt!

6.4 Data fetchen

We weten nu dat we door middel van een specifieke url gegevens kunnen oproepen over het weer: we gaan een GET-request maken naar OpenWeather. Voeg Axios toe als dependency door het volgende in de terminal te typen:

```
npm install axios
```

Nu hoeven we het alleen nog te importeren bovenaan in ons App.js bestand:

```
import axios from 'axios';
```

Nu kunnen we de variable `axios` gebruiken als basis om de verschillende request-methodes die deze library beschikbaar heeft gesteld, te gebruiken. Omdat wij weer voorstellen gaan ophalen, hebben we hier een GET-request voor nodig.

- Schrijf een asynchrone functie `fetchData()` met daarin een `try/catch` blok.
- Maak in het `try` block een GET request naar de OpenWeatherMap API om het weer voor Utrecht op te vragen. Gebruik de GET link uit de vorige paragraaf en vervang de `(NAME STAD)` door `(30ml API KEY)` door jouw `apiKey` variabele. Tip: met string interpolation is dit makkelijker!
- Deze code wordt nog niet afgewerkt (dus er zal ook nog iets gebuurd).

Als het goed is, ziet jouw code er nu zo uit:

```
import React from 'react';
import axios from 'axios';
const apiKey = '085e6262984729404acc4eb353f4';

function App() {
  async function fetchData() {
    try {
      const result = await axios.get(`https://api.openweathermap.org/data/2.5/weather?q=utrecht,nl&appid=${apiKey}&lang=nl`);
      console.log(result.data);
    } catch (e) {
      console.error(e);
    }
  }
}

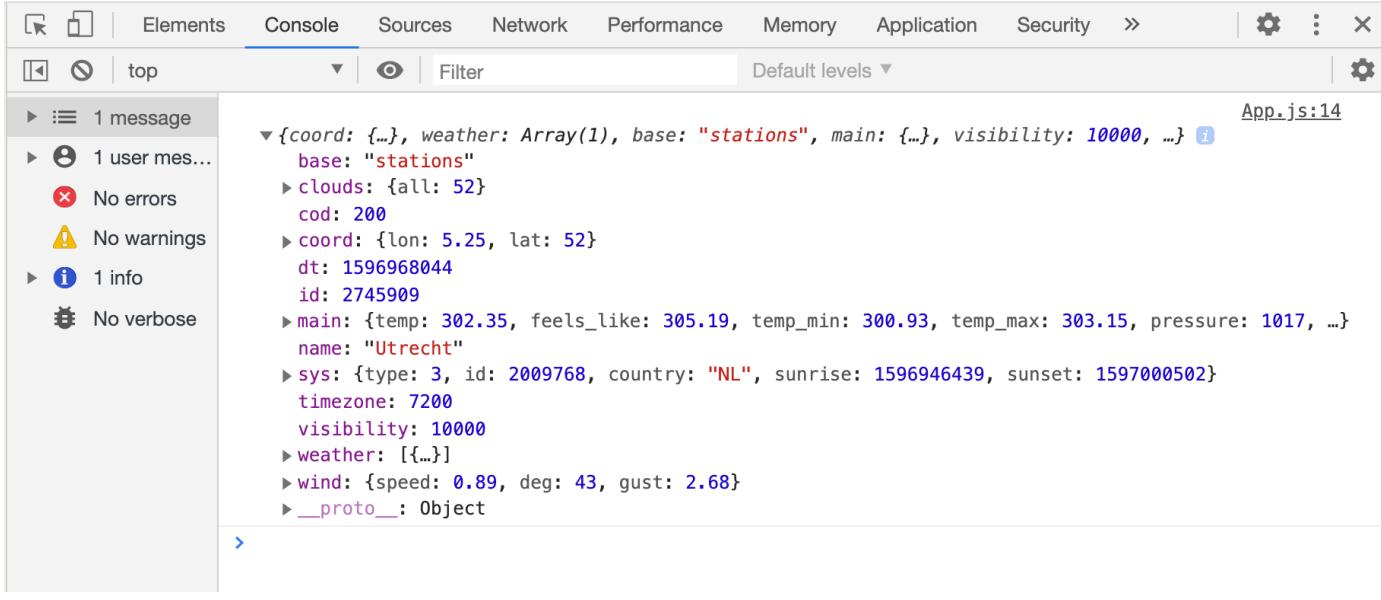
export default App;
```

Ze je hoe we string interpolation hebben gebruikt in de API url en zo de variabele `apiKey` erin gezet hebben? Dat zullen we later ook met de naam van de stad doen, maar voor nu laten we dit hardcoded Utrecht. De data die we terugkrijgen, loggen we nu in de console, maar ook dit gaan we straks aanpakken. Om dit request te triggeren, willen we dat de functie `fetchData()` wordt aangeroepen als de gebruiker op de "Haal data op!" knop drukt.

- Plaats een event listener op de knop en zorg dat de `fetchData()`-functie wordt aangeroepen wanneer de button wordt aangeslagen:

```
function App() {
  return (
    <button
      type="button"
      onClick={fetchData}
    > Haal data op!
    </button>
  );
}
```

Probeer het maar eens. Wanneer je de console openst in Google Chrome (Windows: `Ctrl + Shift + J`, Mac: `option + CMD + J`) en dan op de knop drukt, zie je daar een data object verschijnen:



We moeten de response eerst in de state plaatsen. Als we dat niet zouden doen, blijft de response "gevangen" binnen de scope van de `fetchData`-functie.

- Importeer de `useState`-hook bovenaan het bestand tussen accolades (`import React, { useState } from 'react'`);
- Creeer vervolgens de state variabele `weatherData` en een bijbehorende setter-functie;
- Roep na de `console.log` de setterfunctie aan en geef daar de response mee.

Dan heb je nu dit in jouw editor staan:

```
function App() {
  const [weatherData, setWeatherData] = useState({});

  async function fetchData() {
    try {
      const result = await axios.get(`https://api.openweathermap.org/data/2.5/weather?q=utrecht,nl`);
      setWeatherData(result.data);
    } catch (e) {
      console.error(e);
    }
  }
}
```

Super, nu hebben we de variabele `weatherData` in het gehele `<App>`-component beschikbaar. Laten we de statische tekst in de header vervangen door echte data:

Bekijk het data-object dat in de console staat en ga op zoek naar:

- De naam van de stad
- De beschrijving van het huidige weer
- De temperatuur (dit wordt meegestuurd in de eenheid Kelvin)

Heb je ze gevonden? En tukt het je ook om ze individueel in de console te loggen? Dan kunnen we dat nu in het component plaatsen door de juiste object-keys aan te spreken. En omdat het variabelen zijn, gebruiken we natuurlijk `:`:

```
function App() {
  return (
    <span>{`<h1>${weatherData.name}</h1>
      <h2>${weatherData.weather[0].description}</h2>
      <h3>${weatherData.name}</h3>
      <h1>${weatherData.main.temp}</h1>
      <button type="button" onClick={fetchData}>
        Haal data op!
      </button>
    </span>
  );
}
```

Oeps! Nu krijg je waarschijnlijk enorm veel foutmeldingen als je de applicatie bekijkt, waaronder:

```
TypeError: Cannot read property 'weather' of undefined
```

Dat is gek! Het hele idee van een asynchrone functie was juist dat de functie wacht met het uitvoeren van de codeerna, tot de data binnen is, toch? En dat klopt ook. Alleen halen wij onze data pas op wanneer er op de knop geklikt wordt, terwijl het component eerst volledig gerenderd wordt (zoals er überhaupt een knop is om op te klikken). Tijdens die eerste render is `weatherData` nog een leeg object, en daar zit geen property `weather` op!

Om dit op te lossen, gaan we zorgen dat de elementen die die data nodig hebben alleen worden gerenderd als we ook daadwerkelijk data hebben. Als `weatherData` een primitive datatype was (`string, boolean, number, null, undefined`) konden we simpelweg een impliciete check om de elementen heen zetten:

```
(*Als weatherData een primitief datatype was*)
/weatherData &&
  <>
    <h2>{weatherData.weather[0].description}</h2>
    <h3>{weatherData.name}</h3>
    <h1>{weatherData.main.temp}</h1>
  </>
}
```

Maar omdat wij weatherData initialiseren als leeg object (const { weatherData, setWeatherData } = useState({})); zal dit in bovenstaande check evakuieren naar falsy en krijgen we alsnog een foutmelding. Wanneer we structuurle datatypes gebruiken (zoals objecten en arrays) zullen we moeten checken of er wel keys of waarden in staan, voor we elementen renderen:

```
function App() {
  return (
    <span class="location-details">
      {[Object.keys(weatherData).length > 0 &&
        <>
        <h2>{weatherData.weather[0].description}</h2>
        <h3>{weatherData.name}</h3>
        <h1>{weatherData.main.temp}</h1>
      </>
    </span>
  );
}
```

Omdat we niet één, maar drie elementen willen weergeven wanneer onze weatherData binnen is, hebben we er nog een fragment omheen geplaatst. Je kunt immers maar één component returnen. Nu hebben we geen foutmeldingen meer in onze console en wordt de data weergegeven in onze applicatie nadat we op de knop hebben geklikt.

Omdat de temperatuur in Kelvin wordt aangeleverd (bij het nu alsof het 304.05 graden is in Utrecht... Warmt Dit gaan we later converteren. In de volgende paragraaf gaan we de zoekbalk implementeren, zodat de gebruiker ook naar andere plaatsnamen kan zoeken. Wel zo handig!

Zoek een stad in Nederland

Zoek

Half Bewolkt

Utrecht

304.05

Haal data op!

Vandaag

Komende week

Alle inhoud van de tabbladen komt hier!

Weergeven in C °  ° F

Je kunt hier bekijken hoe [App.js](#) er nu uit zou moeten zien. Bekijk je de stappen lever één voor één? Neem dan een kijkje in de [github](#).

6.5 Callback props

Laten we eens in het `<Searchbar>`-component gaan kijken (`Searchbar.js`). Zodra de gebruiker een zoekterm heeft ingevuld, willen we deze in het parent component (`<App>`) kunnen gebruiken om de weersvoorspelling op te halen. Voordat we dit kunnen doen, zullen we het inputelement om moeten zetten naar een controlled component.

- Importeer `useState` bovenaan het bestand.
- Maké een state aan voor de variabele `query` (een zoekterm noemt men een search query in het Engels) en wijs je een lege string als initiale waarde toe.
- Voeg een event listener toe aan het input element (`onchange`) en zorg dat er bij elke verandering de waarde van `e.target.value` wordt gebruikt om de waarde op te slaan in de `query` variabele.
- Voeg een value-attribuut toe aan het input element toe en geef die de waarde van `query` mee.

Als het goed is ziet jouw code er nu zo uit:

```
function SearchBar() {
  const [query, setQuery] = useState('');

  return (
    <span className="searchbar">
      <input
        type="text"
        name="search"
        value={query}
        onChange={(e) => setQuery(e.target.value)}
        placeholder="zoek een stad in Nederland"
      />
      <button type="button">
        Zoek
      </button>
    </span>
  );
}
```

Op het moment dat de gebruiker op de zoekknop of op Enter drukt, willen we dus iets gaan doen met de query.

- Verberg het span-element voor het formulier.
- Verander de button van type="button" naar type="submit".
- Schrijf een functie genaamd `onFormSubmit`.
- Zorg ervoor dat deze functie wordt aangeroepen wanneer het formulier verzonden wordt.
- Zorg er ook voor dat de pagina niet refresh wanneer dit gebeurt, door gebruik te maken van `e.preventDefault()`.

Super! Op het moment dat deze code wordt uitgevoerd, kunnen we er vanuit gaan dat query een lokale waarde is (anders drukt de gebruiker niet op de knop). Die waarde is op dat moment echter alleen beschikbaar in de state van `<Searchbar>`, en niet in die van `<App>`... En daar hebben we hem nou juist nodig, want daar wordt de data opgehaald...

We hebben dus een manier nodig om vanuit `<Searchbar>` een stukje stats van `<App>` aan te passen. Best ingewikkeld, want properties kun je alleen naar beneden doorgeven (aan children), maar niet terug naar boven (aan parents). Maar wat we wel kunnen doen, is de `funcie` die de state in `<App>` aanpast, doorgeven aan `<Searchbar>`? Dat doen we via callback prop!

Dit is duidelijker als je het ziet. Laten we een stukje state maken in `App.js` waar we straks de lokale in opslaan:

```
function App() {
  const [location, setLocation] = useState('');

  return (
    // ... de rest
  );
}
```

Dan kunnen we die `setLocation` functie nu in `2` in geheide als propste meegeven aan ons `<Searchbar>` component:

```
function App() {
  const [location, setLocation] = useState('');

  return (
    <SearchBar setLocationHandler={setLocation}>
  );
}
```

Het `<Searchbar>`-component kan nu onze `setLocation` functie ontvangen onder de property-naam `setLocationHandler`. Deze property-naam mag je gewoon zelf verzinnen, dus je had hem ook `locationSetter` of - iets minder geschikt - `aardappelSchijfjes` mogen noemen. Laten we ervoor zorgen dat we deze functie nu kunnen gebruiken in ons `<Searchbar>`-component:

- Voeg de parameter `props` toe aan het `Searchbar` component (`<Searchbar.js`) zodat we properties kunnen ontvangen - of destructure `setLocationHandler` al direct.

Je kunt de `setLocation` functie nu gebruiken middels `props.setLocationHandler()` of `setLocationHandler` als je deze gedestructureerd hebt.

- Roope deze functie dus in de ontsnapeert functie die je zojuist gemaakt hebt en geef `query` mee als argument. We willen de `state-setter` functie immers gebruiken om de waarde uit het invoerveld in op te slaan.

Zet jouw `Searchbar`-component er nu zo uit?

```
function SearchBar({ setLocationHandler }) {
  const [query, setQuery] = useState('');

  function onFormSubmit(e) {
    e.preventDefault();
    setLocationHandler(query);
  }

  return (
    <form className="searchbar" onSubmit={onFormSubmit}>
      {/*Inhoud formulier...*/}
    </form>
  );
}
```

Super! Nu krijgt `App.js` telkens een nieuwe lokale beschikbaar in de state als de gebruiker in `<Searchbar>` op 'zoeken' klikt of op enter drukt. Deze lokale zullen we straks gebruiken om de API call te maken naar OpenWeather. Dit begint al op een echte app te lijken! Straks willen we ervoor zorgen dat de functie `FetchData()` in ons `<App>`-component niet wordt aangespoeld wanneer de gebruiker op de 'haal data op' knop drukt, maar wanneer de variabele `location` veranderd is. Maar om dit te kunnen doen, hebben we eerst meer inzicht nodig in de life cycles van React componenten.

Wil je de stappen één voor één zien? Bekijk dan de [github](#) van het `<Searchbar>`-component, of ga direct naar het eindresultaat van [SearchBar.js of App.js](#).

7. Component Life Cycles

7.1 De fases van een component

Eik React component volgt een cyclus vanaf het moment dat het wordt gecreëerd en in de DOM wordt toegevoegd, tot het moment waarop het weer uit de DOM wordt gehaald en vernietigd. Dit is wat we de **Component Life Cycle** noemen. Eigenlijk een beetje zoals een mensleven, als je er zo over nadenk.

Je hebt immiddels geleerd dat wanneer state-variabelen geüpdate worden, deze updates ook gelijk worden doorgevoerd in de UI.

Dit komt omdat componenten opnieuw gerenderd (in de DOM gezet) worden als de bijbehorende state is aangepast.

Dat is natuurlijk handig! Het zorgt er echter ook voor dat acties die slechts één keer (of alleen in specifieke situaties) uitgevoerd mogen worden, niet zomaar in het component gezet kunnen worden. Alles wat we in het component zetten, wordt na iedere re-render opnieuw uitgevoerd:

```
function ExampleComponent() {
  const array = new Array(1, 2, 3); // <---- variabele is direct op het component gedeclareerd
  return (
    //...
  );
}
```

In veel gevallen is dat geen probleem. Maar soms kan het onnodig vaak aanroepen van code een hoop bugs veroorzaken. Daarom biedt React de `useEffect`-hook, waardoor we kunnen bepalen in welke fase van de cyclus onze code moet worden uitgevoerd.

Wanneer gaan we dit gebruiken?

- Wanneer we externe data ophalen. Dit willen we eerstmaal doen, namelijk zodra de gebruiker een pagina open. Wanneer we de data binnen hebben, willen we niet dat de functie nogmaals wordt uitgevoerd.
- Als een component niet meer weergegeven wordt, wil je ook dat de onderliggende processen stoppen. Zoals intervallen (met de `setInterval`-functie van JavaScript) of timers (met de `setTimeout`-functie) die je eerder hebt aangewend of uitgaande netwerk requests die nog niet klaar waren.
- Wiellicht heb je grafische elementen in een canvas `<canvas>`-gegenerer, en moet dit in real-time aangepast worden zodra we andere waarden krijgen.
- Het komt vaak voor dat we data willen ophalen of veranderen, maar pas wanneer een specifieke waarde in de state geüpdate is.

Wanneer gebruik je de `useEffect`-hook? We kunnen de verschillende fases van de cyclus categoriseren in 3 (oké, 3 en een beetje) stadia:

1. Initialisatie: dit wordt aangesloten, maar al dat niet met ons te maken heeft.
2. Mounting (het component blijven in de DOM, chek: renderen of de pagina).
3. Updating (wanneer de properties veranderen van het component aangepast wordt)
4. Unmounting (het component uit de DOM verwijderen en vernietigen)

De `useEffect`-hook zorgt ervoor dat we specifiek gedrag (een side effect) kunnen triggeren op een specifiek moment. We kunnen de `useEffect`-hook dus gebruiken na de mounting-, updating- en unmounting fase. Hoe we dit precies doen, leer je in de volgende paragrafen.

7.2 De mounting cycle

De mounting cyclus is het moment nadat ons component voor het eerst is gerenderd. Denk hierbij weer aan het voorbeeld van de productpagina voor griesmeelpudding van Albert Heijn. De gebruiker vraagt de productpagina op door naar www.ah.nl/product/285837 te navigeren, en alle componenten (de ingrediëntenlijst, afbeelding, prijs en algemene productinformatie) moeten in de DOM geplaatst worden. Op dit specifieke moment moet er echter ook data opgehaald worden, namelijk: de gegevens over griesmeelpudding die we aan al onze componenten willen doorgeven. Om er voor te zorgen dat dit geburt wanneer de pagina wordt geladen (en alleen dan) gebruiken we de `useEffect`-functie van React.

Dit heeft de volgende vorm:

```
useEffect(() => {
  // ...
}, []);
```

Dit tweede argument, de dependency array, is erg belangrijk voor de werking van `useEffect`. We onderscheiden hierin drie opties:

- Een lege dependency array []: de callback wordt uitgevoerd na mounting.
- Een dependency array met één of meerdere waarden: [state varLabel, propertyLabel]: de callback wordt uitgevoerd nadat één van de meegegeven waarden is veranderd, na update.
- Geen dependency array: de callback wordt uitgevoerd op elke re-render (spans hele fase).

Het ophalen van data willen we meestal slechts één keer uitvoeren. In dit geval geven we de `useEffect`-hook dus altijd een lege dependency-array mee, zodat de data na het mounten van het component wordt opgehaald. Deze functie komt altijd buiten de `return` statement te staan, maar wel binnen het component:

```
function ProductPage() {
  useEffect(() => {
    console.log('Ik zie mij alleen op refresh');
  }, []);
}
```

7.3 De update cycle

De update cycle zorgt ervoor dat we acties kunnen uitvoeren die afhankelijk zijn van het updaten van andere waarden. De code die wij dan willen uitvoeren, zou je een soort side effect noemen. Neem bijvoorbeeld dezelfde productpagina, waar twee buttons op staan die het aantal voor count1 en count2 kunnen ophogen:

```
function ProductPage() {
  const [count1, setCount1] = useState(0);
  const [count2, setCount2] = useState(0);

  return (
    <>
      <button
        type="button"
        onClick={() => setCount1(count1 + 1)}
      > Verhoog count 1
      </button>
      <button
        type="button"
        onClick={() => setCount2(count2 + 1)}
      > Verhoog count 2
      </button>
    </>
  );
}
```

Misschien willen we alleen een actie uitvoeren als de hoeveelheid van count2 is veranderd. In dit geval geven we dan count2 mee als afhankelijkheid in de dependency array:

```
function ProductPage() {
  const [count1, setCount1] = useState(0);
  const [count2, setCount2] = useState(0);

  useEffect(() => {
    console.log('count2 is veranderd');
  }, [count2]); // <---- count2 is een afhankelijkheid

  return (
    <>
      <button
        type="button"
        onClick={() => setCount1(count1 + 1)}
      > Verhoog count 1
      </button>
      <button
        type="button"
        onClick={() => setCount2(count2 + 1)}
      > Verhoog count 2
      </button>
    </>
  );
}
```

Dit specifieke `useEffect`-functie wordt iedere keer afgevoerd nadat de waarde in count2 is aangepast.

Side note: wanneer dit component voor het eerst is gerenderd, zal je de boodschap "count2 is veranderd" al in de console zien verschijnen, ondanks het feit dat je nog op geen enkele knop hebt gedrukt. Dit komt omdat het aanmaken van de state, waarin de waarde van count2 van "niks" naar haar initiele waarde (0) wordt veranderd, ook telt als een update.

7.4 De unmount cycle

Als een component niet meer weergegeven wordt, wil je ook dat de onderliggende processen stoppen, zoals timers, intervallen of uitgaande netwerk requests die nog niet klaar waren. De `useEffect`-functie is zo geschreven dat we een `cleanup`-functie mogen terugvoeren uit de callback. Alles wat in deze cleanup-functie gedefinieerd wordt, zal uitgevoerd worden nadat het component ge-unmount is:

```
useEffect(() => {
  // acties die uitgevoerd worden na mount
  return function cleanup() {
    // acties die uitgevoerd worden na unmount
  };
}, []);
```

Netwerk requests opruimen

We gebruiken de `cleanUp`-functie om lopende requests te stoppen, mocht de gebruiker in de tussentijd inens naar een andere pagina binnen de applicatie navigeren. Zo voorkomen we een **memory leak**. Een memory leak vindt plaats wanneer een component ge-unmount wordt; tenslotte de asynchrone functie nog aan het wachten is op de response van de server. Ondanks het feit dat het component al uit de DOM verwijderd is, zal het request nog steeds afgehandeld worden zodra de response ontvangen wordt. De state-setter functie die de ontvangen data in de state zet, wordt vervolgens ook nog aangeroepen. Hierdoor ontstaat de volgende foutmelding:

"Can't perform a React state update on an unmounted component. This problem is due to a common mistake: using hooks inside a component's constructor, or inside a function component's body. Instead, move such logic to a regular function component."

Het probleem zit 'm hier niet in het feit dat de state-setter functie wordt aangeroepen. Het probleem is dat er een hoop werkgeheugen in beslag genomen wordt door processen die niet eens meer relevant zijn voor de gebruiker. De applicatie crasht dus niet, maar wordt op ten duur zeer traag.

Om onze requests in dit geval te kunnen annuleren, hebben we een actie `abortcontroller` nodig. Eerst maken we met actie een abortcontroller object aan:

```
const controller = new AbortController();
```

De signal key die we op dit object vinden, moet bij ieder request worden meegegeven in het configuratie-object onder de naam `signal`:

```
function ProductPage() {
  useEffect(() => {
    const controller = new AbortController();
    async function getData() {
      const response = await axios.get('https://link-naar-endpoint.nl', {
        signal: controller.signal,
      });
    }
    getData();
  }, []);
  return (
    //...
  );
}
```

*Let op: in dit voorbeeld is het try/catch blok weggelegd om de illustratie simpeler te houden.

No kunnen we dit request vervolgens annuleren in de `cleanup`-functie, door de `axios.abort`-methode aan te roepen:

```
function ProductPage() {
  useEffect(() => {
    const controller = new AbortController();
    async function getData() {
      const response = await axios.get('https://link-naar-endpoint.nl', {
        signal: controller.signal,
      });
    }
    getData();
    return function cleanup() {
      controller.abort(); // <--- request annuleren
    };
  }, []);
  return (
    //...
  );
}
```

Intervalen oproemen

JavaScript ondersteunt twee soorten timing-functies: `setInterval` (om iedere x seconden iets uit te voeren) en `setTimeout` (om x seconden te wachten tot iets uitgevoerd mag worden). Stel dat wij iedere seconde het woordje "loading..." in de console willen weergeven. Deze `setInterval`-functie willen we recht op het component plaatsen, want dan zal er iedere re-render (wanneer bijvoorbeeld de state geüpdatet wordt) een extra interval gestart worden naast alle huidige intervalen die nog bezig waren. We willen deze functie enkel starten, namelijk nadat het component voor het eerst `mounted` is. Dat doen we als volgt:

```
function ProductPage() {
  useEffect(() => {
    setInterval(() => {
      console.log("loading...");
    }, 1000);
  }, []);
  return (
    <p>Open de browserconsole om de intervallen te zien!</p>
  );
}
```

We willen echter ook voor zorgen dat wanneer dit component van deze pagina niet meer actief is, dat ook het interval stoppt. Als we dit niet expliciet benoemen, zal het interval namelijk gewoon doorgaan en creëert dit een memory-leak in onze applicatie. Als we dit willen implementeren voor ons interval, kunnen we daar de ingebouwde `clearInterval`-functie van JavaScript gebruiken die we aanroepen in onze `cleanup`-functie.

```
function ProductPage() {
  useEffect(() => {
    const loop = setInterval(() => {
      console.log("loading...");
    }, 1000);
    return function cleanup() {
      console.log("Het interval wordt gestopt!");
      clearInterval(loop);
    };
  }, []);
  return (
    <p>Open de browserconsole om de intervallen te zien!</p>
  );
}
```

7.5 useEffect in de praktijk

Laten we eens weetvoorziening-applicatie verder uitbouwen. In het vorige hoofdstuk hebben we ervoor gezorgd dat wanneer de gebruiker naar een locatie zoekt, de state variabele `location` in `App.js` wordt veranderd. En wanneer dit gebeurt, willen wij ervoor zorgen dat we data gaan ophalen vanuit de API: onze `fetchData`-functie moet worden uitgevoerd.

- Houd er bij de beginnen de ogen open dat je `useEffect`-hook maar weg uit de header. We plan nemelijk de `useEffect`-hook gebruiken!
- Plaats de `useEffect`-hook nu in de `App`-component (een `React`-component) en niet in de `weatherData`-component.
- Plaats de `useEffect` functie in het `App`-component onder de state-initialisatie. Deze functie krijgt twee parameters:

- Een callback functie (met de code die uitgevoerd wordt)

- De dependency array (zodat we weten wanneer het uitgevoerd moet worden)

Als we de data willen ophalen wanneer de state-variabele `location` is veranderd, welke versie van de `useEffect`-hook hebben we dan nodig?

...

...

De update cycle! In ons geval is de dependency onze state variabele `location`: we willen immers dat de data **alleen** wordt opgehaald wanneer de gebruiker naar een locatie zoekt. En op het moment dat deze locatie veranderd is (van een lege string naar 'utrecht' of van 'utrecht' naar 'amsterdam'), dan moet de `fetchData()` functie aangeroepen worden. Dat heb je als het goed is op deze manier gedaan:

```
function App() {
  useEffect(() => {
    // 1. we definiëren de functie
    async function fetchData() {
      try {
        const response = await axios.get(`https://api.openweathermap.org/data/2.5/weather?q=utrecht,nl&appid=${apiKey}&lang=nl`);
        setWeatherData(result.data);
      } catch (e) {
        console.error(e);
      }
    };
    // 2. we roepen de functie aan
    fetchData();
    // code wordt alleen afgevuurd als location veranderd
  }, [location]);
  return (
    //...
  );
}
```

Let op: Als je vergeet de dependency-array mee te geven, creëert je een oneindige loop. Iedere keer als de waarde van de state (`weatherData`) in de asynchrone-functie wordt aangepast, wordt het component opnieuw gerenderd, waarna de `useEffect`-hook weer wordt aangeroepen, waarna de state weer wordt aangepast, etc. Uiteindelijk crasht je applicatie omdat de elementen van dit component te vaak moeten re-renderen.

Op dit moment wordt de `fetchData`-functie echter ook aangeroepen in de initialisatie-fase. Wanneer de state variabele `location` van "niet bestaand" naar een lege string '' wordt veranderd bij het aannemen van de state, zal de `useEffect`-hook ook getriggert worden. We hebben daarom een extra check nodig.

- Plaats een `if`-statement om de aanroep van de `fetchData`-functie heen, die checkt of `location` wel een `truthy` waarde (een gevulde string) heeft

```
function App() {
  useEffect(() => {
    // 1. we definiëren de functie
    async function fetchData() {
      try {
        const response = await axios.get(`https://api...`);
        setWeatherData(result.data);
      } catch (e) {
        console.error(e);
      }
    };
    // 2. we roepen de functie aan als location is veranderd,
    // maar niet als het een null/undefined/lege string is
    if (location) {
      fetchData();
    }
  }, [location]);
  return (
    //...
  );
}
```

Superf! Nu moeten we onze API url nog aanpassen zodat we een dataverzoek doen naar de stad waar de gebruiker naar heeft gezocht.

```
axios.get(`https://api.openweathermap.org/data/2.5/weather?q=${location},nl&appid=${apiKey}&lang=nl`);
```

Hij doet het Probeer het zelf maar eens als je zoekt op Utrecht, Amsterdam, of vanuit welk plek in Nederland jij nu aan het studeren bent!

Wil je de stappen één voor één zien? Bekijk dan de [gitHistory](#) van het `<App />`-component, of ga direct naar het eindresultaat van [App.js](#).

7.6 useEffect in de praktijk continued

Laten we nu de voorspelling voor komende week eens functionele maken!

- Importeer in App.js het `<ForecastTab />`-component en plaats hem op de plek waar nu "Alle inhoud van de tabbladen komt hier!" staat.

Dit ziet er zo uit:

```
import ForecastTab from './pages/forecastTab/ForecastTab';

function App() {
  return (
    <div className="weather-content">
      <TabBarMenu/>
      <div className="tab-wrapper">
        <ForecastTab />
      </div>
    </div>
  );
}
```

Als het goed is, zie je nu vijf keer de voorspelling voor maandag, waarop het 12 graden wordt met lichtelijke bewolking. Bovendien zal je in het `<ForecastTab />`-component zien dat dezelfde code vijf keer onder elkaar gekopieerd is. Dit willen we natuurlijk anders doen!

We gaan een nieuw request doen naar de OpenWeather API om zo de weervoorspellingen voor komende week op te vragen. Hiervoor sturen we niet de plaatsnaam mee, maar de coördinaten van de plek waarover we de voorspelling willen ontvangen. Dat heeft de maker van de API zo vastgesteld. Gelukkig hoeven we deze coördinaten niet zelf te bepalen, want deze informatie hebben we al teruggekregen in de response van het vorige request. We hoeven dus alleen het `weatherData`-object in de state aan te spreken, onder de key `coord`:

- Geef deze coördinaten aan ons `<ForecastTab />`-component, onder de property-naam `coordinates`

Als het goed is, ziet jouw code er nu zo uit:

```
import ForecastTab from './pages/forecastTab/ForecastTab';

function App() {
  return (
    <div className="tab-wrapper">
      <ForecastTab coordinates={weatherData.coord}/>
    </div>
  );
}
```

Goed, laten we eens een kijkje nemen in `ForecastTab.js`. We gaan hier ook een API call maken, maar dit keer naar een ander endpoint. Volgens de documentatie van OpenWeather hebben we de volgende url nodig om een dagelijks weervoorspelling terug te krijgen:

```
https://api.openweathermap.org/data/2.5/forecast?lat=\${VOEG\_HIER\_LATITUDE\_TOE}&lon=\${VOEG\_HIER\_LONGITUDE\_TOE}&appid=\${JOOUW\_API\_KEY}&lang=nl
```

Vooraleer we een request kunnen maken naar dit endpoint, zullen we eerst een aantal kleine dingen moeten implementeren:

- Zorg ervoor dat de property `coordinates` ook beschikbaar is in het `<ForecastTab />`-component. Mocht je willen kijken hoe ons `coordinates`-object eruit ziet, kun je hem in een `console.log()` in het `<ForecastTab />`-component plaatsen. Uiteraard verschilt hij pas als je via de interface naar een plaatsnaam hebt gezocht!
- Maké een nieuwe state aan voor de variabele `forecast` en zet de initiale waarde hierop op `[]`.
- Importeer de `useEffect`-hook en zorg ervoor dat deze hook wordt aangeroepen zodra de property `coordinates` verandert. Tip: `coordinates` hoort dus in de dependency array!
- Importeer `axios` bovenaan in het bestand zodat je erals een GET-request kunt maken.
- Declarer een asynchrone functie binnen de `useEffect`-hook en noem deze `fetchWeather`. (Je kunt een kegels spelen in App.js). Deze functie heeft een `try/catch` blok nodig.
- Declarer de variabele `response` uit App.js en zorg ervoor dat deze ook in de `useEffect`-hook is. Je kunt hier de status toevoegen.
- Maké een nieuwe GET-request die de `coordinates` verwent. Vul de juiste informatie in op de lege plekken van de url. Tip: Het kan heel goed zijn dat WebStorm de object keys in het `coordinates` object niet goed herkent. Dus maak je geen zorgen als daar een melding onder staat. De response value van het GET-request mag je eerst even loggen in de console, doormiddel van `console.log(response.data)`.
- De `fetchWeather`-functie wordt nu nog niet aangeroepen. Roep de functie aan in de `useEffect`-hook, maar zorg ervoor dat dit alleen gebeurt wanneer er ook daadwerkelijk coördinaten aanwezig zijn. Als de gebruiker nog niet naar een plaatsnaam heeft gezocht, komt `coordinates` binnen als `undefined`. Hier kun je dus een impliciete check voor gebruiken.

Zo! Dat was een hoop code. Je kunt de stappen tot het eindresultaat in `ForecastTab.js` nog eens rustig doorlopen via de [gitHistory](#), of direct het eindresultaat van `ForecastTab.js` en `App.js` bekijken.

7.7 Data iteratie en keys

Als we nu de response `value` bekijken in de console, zien we het volgende:

Elements Console Sources Network > 1 | | |

top | Filter Default levels |

1 Issue: 1

ForecastTab.js:14

```
▼ {cod: '200', message: 0, cnt: 40, list: Array(40), city: {...} } ⓘ
  ► city: {id: 2745909, name: 'Utrecht', coord: {...}, country: 'NL', pop
    cnt: 40
    cod: "200"
  ▼ list: Array(40)
    ► 0: {dt: 1669744800, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 1: {dt: 1669755600, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 2: {dt: 1669766400, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 3: {dt: 1669777200, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 4: {dt: 1669788000, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 5: {dt: 1669798800, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 6: {dt: 1669809600, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 7: {dt: 1669820400, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 8: {dt: 1669831200, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 9: {dt: 1669842000, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 10: {dt: 1669852800, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 11: {dt: 1669863600, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 12: {dt: 1669874400, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 13: {dt: 1669885200, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 14: {dt: 1669896000, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 15: {dt: 1669906800, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 16: {dt: 1669917600, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 17: {dt: 1669928400, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 18: {dt: 1669939200, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 19: {dt: 1669950000, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 20: {dt: 1669960800, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 21: {dt: 1669971600, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 22: {dt: 1669982400, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 23: {dt: 1669993200, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 24: {dt: 1670004000, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 25: {dt: 1670014800, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 26: {dt: 1670025600, main: {...}, weather: Array(1), clouds: {...}, wi
    ► 27: {dt: 1670036400, main: {...}, weather: Array(1), clouds: {...}, wi
```

In de key `list` vinden we een array met veertig objecten. Op dit moment krijgen we de voorspellingen in blokken van 3 uur binnen, voor de komende vijf dagen. Omdat we hiervoor moeten betalen om een samenvattende voorspelling per dag op te halen, zullen we dit zo goed mogelijk proberen te benaderen door uit deze lijst alleen de voorspellingen van 12 uur's middage te gebruiken.

Klap één zo'n voorspelling-object maar eens open. In de key van ieder object (`dt, txt`) staat de tijd van dat blok.

• Reduzer deze array van veertig objecten tot vijf objecten, op basis van de volgende conditie: In de `dt, txt` property van het object moet de string "12:00:00" voorkomen. Tip: Hiervoor heb je twee ingebouwde JavaScript methoden nodig! Slab deze nieuwe array op in een variabele genaamd `fiveDayForecast`.

• Gebruik vervolgens de state-setter-methode `setForecasts` om deze nieuwe array in de state te plaatsen.

Als het goed is, heb je dit ongeveer op deze manier gedaan:

```
function ForecastTab({ coordinates }) {
  const [forecasts, setForecasts] = useState([]);
  useEffect(() => {
```

```

async function fetchForecasts() {
  try {
    const response = await axios.get('https://...');

    // 1. Filter de resultaten op basis van onze indicatie
    // dat het een forecast is. Resultaat: data.lijn.filter((singleForecast) => {
      return singleForecast.dt.txt.includes("12:00:00");
    });
    // 2. Stop de array van vijf voorspellingen in de state
    setForecasts(fiveDayForecast);
  } catch(e) {
    console.error(e);
  }
}

if(coordinates) {
  fetchForecast();
}

[coordinates];

```

Nu willen we deze voorspellingen natuurlijk laten zien aan de gebruiker. En omdat we een array in de state hebben geplaatst, kunnen we nu over deze array heen itereren met de `map`-methode. Op deze manier kunnen we voor iedere weersvoorspelling in de array een `<article>`-element op de pagina plaatsen:

```

function ForecastTab({ coordinates }) {
  // .. alle state en effects

  return (
    <div className="tab-wrapper">
      {forecasts.map((day) => {
        return (
          // alle componenten die we herhaaldelijk willen teruggeven
        );
      })}
    </div>
  );
}

```

Side note: de reden dat we nu geen impliciete check hoeven te gebruiken voor `forecasts`, is omdat we deze state variable hebben geinitialiseerd als lege array. Wanneer we een loop maken over een lege array, geeft dit geen foutmeldingen. De `map`-methode stopt dan simpelweg direct nadat hij begint.

We welen nu voor elk item in de `forecast`-array een `<article>` classmate "`forecast-day`" toegeven met alle bijbehorende elementen in:

- Elk bestaand `<article>` in de return waarde en verenigt alle andere `<article>`-tags uit het component.
- Vervang de statische temperatuur en weer beschrijving door dynamische waarden.

Om de juiste dag waar te geven hebben we de property `dt` nodig. Die staat voor Date Time. Echter, als je de documentatie van deze API zou lezen, zou je erachter komen dat deze waarde als UNIX timestamp (aantal seconden sinds 1970) wordt meegegeven. *“Die is voor de Java Script opdrachten”*

Schrijf een functie die een timestamp als parameter verwacht en een Nederlandse, geformateerde `weekdag` teruggeeft. Houd er rekening mee dat je de timestamp met 1000 moet vermengen om er een geldige datum van te maken. Als je even niet meer weet hoe je datums kunt formateren, pak dan hoofdstuk 5.5 (Date object) uit de cursus JavaScript er even bij!

Vervolgens uitkomst:
#606933200 geeft "Dinsdag"
#807079600 geeft "Vrijdag"
#667166000 geeft "Zaterdag"

Komen de JavaScript-afleveringen die je hebt gemaakt toch van pas. Ziet jouw functie er nu ongeveer zo uit?

```

function createDateString(timestamp) {
  const day = new Date(timestamp + 1000);
  return day.toLocaleDateString('nl-NL', { weekday: 'long' });
}

```

Super! Nu kun je deze functie gebruiken om alle timestamps mee te formatteren en ziet jouw code er nu zo uit:

```

function createDateString(timestamp) {
  const day = new Date(timestamp + 1000);
  return day.toLocaleDateString('nl-NL', { weekday: 'long' });
}

function ForecastTab({ coordinates }) {
  // useEffect en state methode staan hier...

```

```

  return (
    <div className="tab-wrapper">
      {forecasts.map((singleForecast) => {
        return <article className="forecast-day">
          <p className="day-description">
            {createDateString(singleForecast.dt)}
          </p>

          <section className="forecast-weather">
            <span>
              {singleForecast.main.temp}&deg; C
            </span>
            <span>
              {singleForecast.weather[0].description}
            </span>
          </section>
        </article>
      ))}
    </div>
  );
}

export default ForecastTab;

```

Op dit moment hebben we wel een kleine kleine error in onze console staan. Dit kunnen we oplossen door het buitenste component dat we `herhalen` (in dit geval ons `<article>`) in de `map`-methode een `key`-attribuut mee te geven. Dit zorgt ervoor dat React niet in de war raakt wanneer er één van de bijna identieke `<article>`-elementen geüpdatet moet worden.

De key moet altijd op het buitenste element staan, en bovendien uniek zijn. Je mag hiervoor niet het index-nummer van de array gebruiken. In ons geval kunnen we onze date timestamps gebruiken, omdat ze altijd uniek zijn:

```

function ForecastTab({ coordinates }) {
  // .. alle state en effects

  return (
    <div className="tab-wrapper">
      {forecasts.map((singleForecast) => {
        return <article className="forecast-day" key={singleForecast.dt}>
          //Alle onderdelen van dit artikel
        </article>
      ))}
    </div>
  );
}

```

En tadaaaal Weersvoorspelling voor de komende vijf dagen. Wil je de stappen één voor één zien? Bekijk dan de [diffstory](#) van het `<ForecastTab />`-component, of ga direct naar het eindresultaat van `ForecastTab.js`.

8. Clean code & best practices

8.1 Gebruikersfeedback: errors

Fouten maken doet iedereen, maar vooral onze gebruikers. Daarom moeten we ervoor zorgen dat we in iedere situatie feedback geven over wat er gebeurt, of wat er dus juist niet gebeurt. Stel, onze gebruiker zoekt naar 'Amsterdam' in plaats van 'Amsterdam'. Die plaatsnaam bestaat niet, dus zal de API een error-code terug geven wanneer het weer voor deze plaatsnaam opvraagt. Dit wordt netjes opgegeven in ons `catch`-blok van het `<App />`-component:

```

// App.js
async function fetchData() {
  try {
    const response = await axios.get('https://api.openweathermap.org...');
    setWeatherData(response.data);
  } catch (e) {
    // de error wordt weergegeven in de console
    console.error(e);
  }
}

```

In dit geval is de error message die we terugkrijgen: 'Request failed with status code 404'. Maar wat betekenen die codes eigenlijk? Laten we even door de meest veelvoorkomende lopen:

- Als de status 200 is, klopt alles goed.
- Als de status 401 is, dan heeft de gebruiker niet de juiste autorisatie om de website te openen of de server heeft niet genoeg.
- 401 Unauthorized: De server begrijpt je verzoek, maar je bent niet geautoriseerd om met dit verzoek te communiceeren (misschien je voorbeeldig ingelogd moet zijn of deze data alleen als admin mag zien). Check in dit geval of je wel duidelijk maakt wat jouw rechten zijn in de request header.
- 403: Forbidden: De server begrijpt je verzoek, maar weigert deze. Dit heeft niets met autorisatie te maken.
- 500 Internal Server Error: Er is een onverwachte fout opgetreden bij de server waardoor het verzoek niet uitgevoerd kan worden. Wat er precies misgaat, wordt in deze generieke foutmelding nooit benoemd.
- 502 en 503: De server is tijdelijk niet beschikbaar.

Er bestaat ook een 419 melding, die betekent: 'I'm a teapot', ja, echt. Dit is het overblijfsel van een 1-april-grap uit 1998...

Wanneer iets misgaat in ons request, wordt de statuscode in het rode weergegeven in de console. Handig voor ontwikkelaars, maar de gemiddelde gebruiker zal de console natuurlijk nooit openen. Bovendien zullen weinig gebruikers weten waar deze codes voor staan. Het zou daarom zeer gebruiksvriendelijk zijn om het bij onze `console.error` te laten. In plaats daarvan willen we een duidelijke melding in de UI plaatsen dat deze plaatsnaam niet bestaat.

- Maak een kleine state aan met de `useState`-hook en noem de variabele `error`, met een initiale waarde van `false`.
- Gebruik de `setError`-functie om `error` op `true` te zetten in het `catch`-blok.
- We moeten ervoor zorgen dat, indien keert voor dat er een request wordt gedaan, `error` eerst op `false` gezet wordt. Het kan namelijk voorkomen dat de vorige zoekopdracht eindigde in een error, waarna we bij deze poging weer starten met een 'schone lei'.

Als het goed is ziet puwe code er nu zo uit:

```
function App() {
  // -- andere state declaraties --
  const [error, toggleError] = useState(false);

  useEffect(() => {
    async function fetchData() {
      try {
        const response = await axios.get('https://api.openweathermap.org/...');

        setWeatherData(response.data);
      } catch (e) {
        console.error(e);
        toggleError(true);
      }
    }

    if (location) {
      fetchData();
    }, [location]);
  });

  return (
    <>
    /* -- alle elementen -- */
    </>
  )
}
```

Na kunnen we `error` gebruiken om een conditionele foutmelding te laten zien onder de zoekbalk:

```
// App.js
<SearchBar setLocationHandler={setLocation}>
  {error &&
    <span className="wrong-location-error">
      Dots! Deze locatie bestaat niet
    </span>
  }

```

Dat ziet er goed uit. Probeer maar eens de zoeken op 'Asterdam' of 'Timboektoe' om de foutmelding te zien verschijnen! Laten we dit ook implementeren in `ForecastTab.js`. Volg de stappen opnieuw:

- Maak een `state` aan voor de `error`-variabele;
- Gebruik de `setError`-functie om de waarde op `true` te zetten in het `catch`-blok;
- Zorg dat we de waarde vóór elk `new request` eerst op `false` zetten;
- Gebruik de waarde van `error` om de melding 'Er is iets misgegaan bij het ophalen van de data' te laten zien.

Zet jouw code er nu zo uit?

```
function ForecastTab({ coordinates }) {
  const [forecasts, setForecasts] = useState([]);
  const [error, toggleError] = useState(false);

  useEffect(() => {
    async function fetchForecasts() {
      try {
        toggleError(false);
        const response = await axios.get('https://api.openweathermap.org/...');

        const fiveDayForecast = response.data.list.filter(singleForecast => {
          return singleForecast.dt_txt.includes('12:00:00');
        });

        setForecasts(fiveDayForecast);
      } catch (e) {
        console.error(e);
        toggleError(true);
      }
    }

    if (coordinates) {
      fetchForecasts();
    }, [coordinates]);
  });

  return (
    <div className="tab-wrapper">
      {error && <span>Er is iets misgegaan met het ophalen van de data</span>}
      {/* Alle andere elementen... */}
    </div>
  );
}
```

We zijn goed bezig! Laten we dan nog één laatste bericht plaatsen voor de gebruiker. Als er nog één weersvoorspelling opgehaald is, willen we 'Zoek eerst een lokale om het weer voor deze week te bekijken' laten zien.

- Laat een `` met bovenstaande melding zien wanneer er één weersvoorspelling (`forecast` heeft als lengte 1) is en niks is misgegaan (`error` is dan `false`).
- Geef die span een class met `'no-forecast'`.

```
{forecasts.length === 0 && !error &&
<span className="no-forecast">
  Zoek eerst een lokale om het weer voor deze week te bekijken
</span>
}
```

Fantastisch, nu hebben we alle mogelijke uitkomsten opgevangen en weet de gebruiker precies waar hij of zij aan toe is! Tip: je zou hier ook een ternary operator voor kunnen gebruiken, om te differentiëren tussen de melding en de weersvoorspelling.

Bekijk hier het eindresultaat voor `App.js`, of volg de stappen één voor één in de `gitHistory`. Het eindresultaat voor `ForecastTab.js` kun je ook bekijken, inclusief de `testcases`.

8.2 Gebruikersfeedback: laadtijd

In Nederland zijn we thuis en op ons werk meestal geopend met 'slow' internet. We moeden echter ook rekening houden met gebruikers die gebruik maken van verbindingen die bijvoorbeeld met hun telefoon op een 3G netwerk zitten. Volgens J. Nielsen (1993) is een wachttijd van meer dan 1 seconde al genoeg om de gebruikservering te verstoren. Het is daarom belangrijk om de gebruiker te laten weten dat er ook waardeloos werk gebeurt wanneer wij een API-call maken en wachten op de response data. Dit kun je bijvoorbeeld doen door een loader-animatie, of simpelweg door het zelfde Data wordt opgehaald te laten zien.

Laten we dit direct toe passen in de voorbereiding voor de komende vijf dagen. Open `ForecastTab.js` en implementeer de volgende stappen:

- Maak een `state` aan en noem de variabele `loading`, met een initiale waarde van `false`;
- Gebruik de `setError`-functie om `loading` op `true` te zetten wanneer we een error krijgen (dat is óók het `try/catch`-blok)
- Gebruik de `toggleError`-functie om `loading` op `false` te zetten als alles is gelukt. Maar zijn met het ophalen van de data (na het `try/catch`-blok)
- Gebruik de waarde van `loading` om de melding 'Loading... - te laten zien tijdens het laden.

Als het goed is ziet jouw code er zo uit:

```
function ForecastTab({ coordinates }) {
  // -- andere state declaraties
  const [loading, toggleLoading] = useState(false);

  useEffect(() => {
    async function fetchForecasts() {
      toggleLoading(true);
      try {
        toggleError(false);
        const response = await axios.get('https://api.openweathermap.org/...');

        const fiveDayForecast = response.data.list.filter(singleForecast => {
          return singleForecast.dt_txt.includes('12:00:00');
        });

        setForecasts(fiveDayForecast);
      } catch (e) {
        console.error(e);
        toggleError(true);
      }
      toggleLoading(false);
    }

    if (coordinates) {
      fetchForecasts();
    }, [coordinates]);
  });

  return (
    <div className="tab-wrapper">
      {loading && <span>Er is iets misgegaan met het ophalen van de data</span>}
      {/* Alle andere elementen... */}
    </div>
  );
}
```

Als jij een snelle verbinding hebt, zal je deze melding niet eens zien, omdat `loading` maar een milliseconde `true` zal zijn. Maar gebruikers met een langzame verbinding zijn hier heel blij!

Bekijk hier het eindresultaat voor `ForecastTab.js`, of volg de stappen één voor één in de `gitHistory`.

8.3 Tabbladen

Het is tijd om ons tweede tabblad functioneel te maken. Deze paragraaf telt niet echt mee als `best practice`, maar is wel nodig om onze applicatie te laten werken! Laten we beginnen met de package React Router. Deze moeten we eerst even installeren in ons project:

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

- Plaats een `Router-component` en `Route`s applicatie in `Index.js`. Verget deze niet eerst te importeren!
- Importeer in `App.js` de `Router` en `Routes`:
- Zet het koude component en `div className="tab-wrapper"`:
- Zorg dat `<ForecastTab>` weergegeven wordt op de url /komende-week

• Zorg dat <TodayTab> weergegeven wordt op de url / Let op: dit component moet nog wel geimporteerd worden.
Dat ziet nu als het goed is zo uit:

```
import { Routes, Route } from 'react-router-dom';
function App() {
  // State declaraties...
  useEffect(() => {
    // data ophalen...
  }, [location]);
  return (
    <>
      {/* Overige elementen... */}
      <div className="tab-wrapper">
        <Routes>
          <Route path="/" element={<TodayTab />} />
          <Route path="/komende-week" element={<ForecastTab coordinates={weatherData.coord}/>} />
        </Routes>
      </div>
    </>
  );
}

Will je het eindresultaat van App.js bekijken? Dat kan hier. Je kunt ook de stappen even naloopen via de oplossing.
```

Laten we nu onze menu-items ook functionalen maken. Open het bestand TabBarMenu.js:

- Vervang alle <a>-elementen voor <NavLink>-componenten (en vergeet deze niet te importeren!)
- Verander de href-attributen naar to-attributen en zorg dat die url's goed staan. Wijst Vandaag naar "/" en Komende week naar "/komende-week".
- Gooien we alleen een CSS-class active hebben, maar geen alternatief, mag je dit met de && operator oplossen:

```
className={({ isActive }) => isActive && 'active'}
```

Bekijk hier het eindresultaat van [TabBarMenu.js](#) of bekijk het stapsgewijs via de [oplossing](#).

8.4 Helperfuncties

Helper-functies zijn JavaScript-functies die een klein hulje voor je opteilen.

Tijd voor een best practice: **code splitting**. We willen onze componenten zo leeg mogelijk houden, dus wanneer we functies gebruiken die geen React gebruiken (zoals state) willen we die het liefst op een aparte plek opslaan. Laten we eens gaan kijken hoe we dit in de WeatherApp voor elkaar kunnen krijgen.

Op dit moment worden alle temperaturen nog in Kelvin weergegeven, maar dat zegt over gedrags natuurlijk niets. We hebben een helper-functie nodig die Kelvin omzet naar Celsius! Je herkent stuksje code die je kunt optellen als helperfuncties, aan het feit dat ze alleen bestaan uit JavaScript. En je schrijft ze vaker dan je denkt! Misschien heb je een functie geschreven die het aantal woorden in een zin uitrekt, of data eerst filtert en daarna sorteert op alfabetische volgorde. Dat soort functie-declaraties willen we tussen de rest van de code te laten staan, omdat het de componenten onnodig "vervuilt". Dus hoe kiezen we dit op?

Zet de jouw er ongeveer zo uit:

```
function kelvinToCelsius(kelvin) {
  return `${Math.round(kelvin - 273.15)}° C`;
}

export default kelvinToCelsius;
```

Of heb je hem misschien nog korter opgeschreven (als naamloze arrow function) omdat de functie alleen iets returned?

```
export default (kelvin) => `${Math.round(kelvin - 273.15)}° C`;
```

Weke optie je ook gekozen hebt, deze kunnen we nu importeren in App.js en gebruiken om de Kelvin-waarden te converteren:

```
import kelvinToCelsius from './helpers/kelvinToCelsius';
function App() {
  render()
  //.., de rest
  <h1>{kelvinToCelsius(weatherData.main.temp)}</h1>
}

Kijk eens aan. We hebben echter ook Kelvin-waarden in ons <ForecastTab>-component, laten we onze functie daar ook gebruiken:
```

```
import kelvinToCelsius from '../../../../../helpers/kelvinToCelsius';
function ForecastTab({ coordinates }) {
  // state declaraties en useEffect
  return (
    <span>
      {kelvinToCelsius(coordinates.main.temp)}
    </span>
  )
}

Maar wacht! Kijk eens goed naar onze createDateString-functie in de ForecastTab-. Deze lukt ook maar een simpel taakje voor ons op en gebruikt geen state... Dus die staat eigenlijk ook voor niets in ons component! Laten we daar ook een helper-functie van maken!
```

- Makkelijk een bestand aan in de helpers-map genaamd `createDateString.js`.
- Verplaats nog de functie naar dit bestand en zorg dat hij default geimporteerd wordt.
- Importeer deze functie vervolgens weer in ForecastTab.js (anders werkt ie niet meer).

```
import createDateString from '../../../../../helpers/createDateString';
function ForecastTab({ coordinates }) {
  // state declaraties en useEffect
  return (
    <p className="day-description">
      {createDateString(coordinates.main.dt)}
    </p>
  )
}
```

Opgemeld staat netjes! Je merkt al dat het erg belangrijk is om tijdens het programmeren alert te zetten op het schrijven van helper-functies. Doeze kur je dan direct verplaatst naar de helpers-map.

We zijn echter nog niet klaar. Het tabblad waarin de uitgedrukte weersvoorspelling voor vandaag willen weergeven (<TodayTab>) is nog een beetje leeg. Ook hier gaan we een GET-request voor doen naar het volgende endpoint:

[https://api.openweathermap.org/data/2.5/weather?lat=\\${VOEG HIER LATITUDE TOE}&lon=\\${VOEG HIER LONGITUDE TOE}&appid=\\${JOUW API KEY}&lang=nl](https://api.openweathermap.org/data/2.5/weather?lat=${VOEG HIER LATITUDE TOE}&lon=${VOEG HIER LONGITUDE TOE}&appid=${JOUW API KEY}&lang=nl)

Tip: Omdat je alle stappen om dat op te halen al eens doorgedaan hebt, kun je ook kijken of het je lukt deze op te halen zonder onderstaande instructies stap-voor-stap te volgen!

- Geef eerst de coördinaten vanuit App.js als properties door aan <TodayTab /> net zoals we gedaan hebben bij <ForecastTab />.
- Plaats de variable `apiKey` met jouw API key bovenaan TodayTab.js:
- Schrijf een asynchrone functie die pas wordt aangeroepen zodra de waarde van de property `coordinates` verandert (dus: tijdens de update cycle):
- Importeer daarmee een Promise en een try/catch blok om daarin de GET-request, en zorg dat je de resultaten logt in de console:
- Vergeet niet de `onMount`-property ook aan de `useEffect` toe te voegen!
- Zorg er ook voor dat errors en de load-momenten in de state oplichten:

```
const apiKey = 'd85e6256cb39361faf2b9daccdfsd4eb33f4';
function TodayTab({ coordinates }) {
  const [error, toggleError] = useState(false);
  const [loading, toggleLoading] = useState(false);

  useEffect(() => {
    async function fetchForecast() {
      toggleError(false);
      toggleLoading(true);

      try {
        const response = await axios.get(`https://api.openweathermap.org/data/2.5/weather?lat=${coordinates.lat}&lon=${coordinates.lon}&appid=${apiKey}&lang=nl`);
        console.log(response.data);
      } catch(e) {
        console.error(e);
        toggleError(true);
      }
    }
    toggleLoading(false);

    if (coordinates) {
      fetchForecast();
    }
  }, [coordinates]);

  return (
    // elementen
  )
}
```

Laten we dan nu even kijken naar de data die we teruggekregen hebben. Dit is een hele grote array met 40 voorspellingen:

The screenshot shows the Chrome DevTools interface with the 'Console' tab selected. A large JSON object is displayed, representing a weather forecast. The object structure includes a root node with 'cod', 'message', 'cnt', 'list', and 'city' fields. The 'list' field is an array containing 40 elements, each representing a single weather forecast entry with 'dt', 'main', 'weather', 'clouds', 'wind', and 'rain' properties. A tooltip indicates there are 1 issue in the console.

```
ForecastTab.js:14
▼ {cod: '200', message: 0, cnt: 40, list: Array(40), city: {...}} ⓘ
  ► city: {id: 2745909, name: 'Utrecht', coord: {...}, country: 'NL', pop: 343000, cod: "200"}
  ▼ list: Array(40)
    ► 0: {dt: 1669744800, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 1: {dt: 1669755600, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 2: {dt: 1669766400, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 3: {dt: 1669777200, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 4: {dt: 1669788000, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 5: {dt: 1669798800, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 6: {dt: 1669809600, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 7: {dt: 1669820400, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 8: {dt: 1669831200, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 9: {dt: 1669842000, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 10: {dt: 1669852800, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 11: {dt: 1669863600, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 12: {dt: 1669874400, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 13: {dt: 1669885200, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 14: {dt: 1669896000, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 15: {dt: 1669906800, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 16: {dt: 1669917600, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 17: {dt: 1669928400, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 18: {dt: 1669939200, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 19: {dt: 1669950000, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 20: {dt: 1669960800, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 21: {dt: 1669971600, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 22: {dt: 1669982400, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 23: {dt: 1669993200, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 24: {dt: 1670004000, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 25: {dt: 1670014800, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 26: {dt: 1670025600, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
    ► 27: {dt: 1670036400, main: {...}, weather: Array(1), clouds: {...}, wind: {...}, rain: null}
```

Nou hoeven we die niet allemaal op te slaan in de state, want we willen slechts de eerste drie momenten. We kunnen daarom de array ophakken met de `slice`-methode, voor we deze in de state zetten.

- Maak een stukje state aan voor de weervoorspellingen;
- Plaats alleen de eerste drie waarden uit de lijst in de state;

Heb je dat als volgt gedaan?

```
setForecasts(response.data.list.slice(0, 3));
```

Hier kunnen we vervolgens heel makkelijk overeen itereren met de map method! We hebben al een <WeatherDetail />-component voor je gemaakt die je herhaaldelijk kunt weergeven op de pagina.

- Importeer dit WeatherDetail-component bovenaan het bestand;
- Map in het <WeatherDetail>-component de classnaam "chart" over de forecasts heen en return een <WeatherDetail />-component voor elke entry in de array;
- Geef deze componenten een unieke id;
- De temperatuur onder de property-naam "temp";
- De globale beschrijving van het weer onder de property-naam "type" (Tip: neem de Engelse weerbeschrijving);
- De gedetailleerde beschrijving van het weer onder de property-naam "description";
- Een key-property met een unieke waarde, zodat React de elementen beter kan onderscheiden. Dit is altijd even zoeken in de data die je hebt, maar in ons geval is de waarde uit forecast.dt sowieso uniek!

Als het goed is, heb je dat op de volgende manier gedaan:

```
<div className="chart">
  {forecasts.map((forecast) => {
    return <WeatherDetail
      key={forecast.dt}
      temp={forecast.main.temp}
      type={forecast.weather[0].main}
      description={forecast.weather[0].description}
    />
  ))}
</div>
```

Wanneer je dit gaat testen, zul je nog steeds de dummy-data (icoontje van het weer) op het scherm zien verschijnen. Dat komt omdat we de data wel hebben doorgegeven aan het <WeatherDetail>-component, maar dit component nog niet zo hebben ingericht dat deze properties ook daadwerkelijk kan ontvangen. Dit maken we in de volgende paragraaf in orde.

Laten we de drie elementen daaronder nu ook vervangen door echte timestamps:

```
<div className="legend">
  {forecasts.map((forecast) => {
    return <span>{forecast.dt}</span>
  ))}
</div>
```

Waarschijnlijk krijg je nu een foutmelding in de console te zien. We zijn vergeten een unieke key toe te voegen! Omdat we bij het <WeatherDetail>-component ook al Forecast.dt gebruiken, is deze key niet meer uniek. Dit kunnen we oplossen met een extra word:

```
return <span key={`${forecast.dt}-timestamp`}>{forecast.dt}</span>
```

- Maak de data timestamp nu echter nog een keer weer meegegeven. Niet erg noodzakelijk voor de gebruiker. We gaan opnieuw een helper-functie schrijven om onze timestamp in 24-uurs notatie weer te geven!
- Maak een nieuwe helper-functie <createTimeString> in de map helpers. Deze werkt een beetje als <createDate>-string, alleen gebruiken we nu een toLocaleTimeString() functie. Weet je even niet meer hoe deze functie ook alweer werkt? Je kunt het in deze [documentatie](#) even nalzen.
- De argumenten die we hieraan meegeven zijn een lege array (omdat de formating die wij gekozen hebben niet land-tijdafhankelijk is) en het optie object { hour: '2-digit', minute: '2-digit' } zodat we de tijden als 09:00 en 11:00 krijgen.

Zet jouw functie er nu uit?

```
function createTimeString(timestamp) {
  const day = new Date(timestamp * 1000);
  return day.toLocaleTimeString([], { hour: '2-digit', minute: '2-digit' });
}

export default createTimeString;
```

- Importeer deze functie in de TodayTab.js en gebruik de functie om de timestamp te formateren.
- Voeg onder de chart een conditionale error-melding en een conditionele loading-melding toe, op basis van onze state variabelen loading en error.

Dan heb je als het goed is dit gedaan:

```
function TodayTab({ coordinates }) {
  // ...effect & state

  return (
    <div className="tab-wrapper">
      <div className="chart">
        {forecasts.map((forecast) => {
          return <WeatherDetail
            key={forecast.dt}
            temp={forecast.main.temp}
            type={forecast.weather[0].main}
            description={forecast.weather[0].description}
          />
        ))}
      </div>
      <div className="legend">
        {forecasts.map((forecast) => {
          return <span key={`${forecast.dt}-timestamp`}>{createTimeString(forecast.dt)}

```

Het is je gelukt! In de volgende paragraaf ga je het <WeatherDetail>-component functioneel maken. Bekijk [hier](#) hoe de gehele applicatie er tot nu toe uit zou moeten zien. Je kunt ook de [TodayTab.js](#) in het bijzonder bekijken, of stap-voor-stap doortrekken met de [github](#).

8.5 Bonus: Component-mapper

In de vorige paragraaf hebben we over de weervoerspelling van de komende paарren gemappt en laten we voor iedere voerspelling een <WeatherDetail />-component zien op de pagina. We geven de properties type, temp en description door, maar we hebben het <WeatherDetail />-component nog niet zo ingericht dat er ook daadwerkelijk iets gedaan wordt met deze data. Laten we daar eens mee beginnen. Open het bestand weatherDetail.js.

- Zorg ervoor dat je deze properties ook daadwerkelijk kunt ontvangen in weatherDetail.js door deze properties te destrueren;
- Vervang de dummy-temp waarde door de property temp en gebruik de helpfunctie kelvinToCelsius om deze waarde om te zetten naar een celsius-rotatie;
- Vervang het statische "Zonig" door de property description;

Zet jouw code er nu zo uit?

```
function WeatherDetail({ description, temp, type }) {
  return (
    <section className="day-part">
      <span className="icon-wrapper">
        <i>icoontje van het weer</i>
      </span>
      <p>{description}</p>
      <p>${kelvinToCelsius(temp)}</p>
    </section>
  );
}
```

Het is tijd voor een aantal afbeeldingen om onze tekst te begeleiden. In dit project vind je in de map icons (in assets) zeven SVG's die het weer representeren. Op basis van het weer type (type) willen we bijpassende icoontjes laten zien. Omdat we talkens een keuze maken op basis van één variabele, gaan we hier een switch statement gebruiken. Hierbij geven we in elke case een waarde terug, maar de SVG afbeelding als component. Doorloop de volgende stappen:

- Maak nog een bestand in het mapje helpers en noem deze iconMapper.js;
- Importeer bovenaan dit bestand React, het standaard statement die je boven alle componenten hebt staan. Daarnaast mag je alle iconen één voor één uit de assets map importen om straks te kunnen returnen uit de switch statement. Weet je niet meer hoe je een SVG gebruikt in React? Neem dan even snel een kijkje in de [documentatie](#).
- Schrijf een functie die een weatherType verwacht. Gebruik deze parameter om op te "switchen" in de switch statement die je in deze functie schrijft. Documentatie over de opbouw van een switch statement vind je [hier](#). De cases waarop je wilt switchen zijn:

 - a. "Sunny": - geef de SVG sun.svg, ivg terug als component
 - b. "Clouds": - geef de SVG clouds.svg, ivg terug als component
 - c. "Drizzle": - geef de SVG sun-rain.svg, ivg terug als component
 - d. "Rain": - geef de SVG rain.svg, ivg terug als component
 - e. "Snow": - geef de SVG snow.svg, ivg terug als component
 - f. "Mist": - geef de "sleutel" en "tang" geven alsmede de SVG wind.svg terug als component

Super, als het goed is ziet jouw iconMapper.js er nu zo uit:

```
import React from 'react';
import { ReactComponent as Sunny } from '../assets/icons/sun.svg';
import { ReactComponent as Rain } from '../assets/icons/rain.svg';
import { ReactComponent as Clouds } from '../assets/icons/clouds.svg';
import { ReactComponent as Drizzle } from '../assets/icons/drizzle.svg';
import { ReactComponent as Wind } from '../assets/icons/wind.svg';
import { ReactComponent as Snow } from '../assets/icons/snow.svg';

function iconMapper(weatherType) {
  switch (weatherType) {
    case 'Clear':
      return <Sun/>;
    case 'Clouds':
      return <Clouds/>;
    case 'Drizzle':
      return <Drizzle/>;
    case 'Rain':
      return <Rain/>;
    case 'Snow':
      return <Snow/>;
    case 'Mist':
      case 'Smoke':
      case 'Fog':
      default:
        return <Wind/>;
  }
}
```

```
export default iconMapper;
```

Later we deze functie nu importen in weatherDetail.js en toepassen. Vervang de woorden *icoontje van het weer* door onze functie waarna we type als argument meegeven. Vergeet niet de () eromheen te zetten!

```
<span className="icon-wrapper">
  {iconMapper(type)}
</span>
```

Onze functie zorgt er nu voor dat er op basis van het weer-type een ander icoontje (SVG component) wordt teruggegeven. En die wordt vervolgens direct gerenderd! Fijne van een cent dus.

Bekijk hier het eindresultaat van WeatherDetail.js of bekijk het stapsgewijs via de [narratie](#).

8.6 Environment variables

We hebben op dit moment op deze plekken een apikey variabele gedeclareerd. Naast het feit dat dit vrij inefficiënt is, is het bovendien ook onveilig.. Je wilt namelijk nooit jouw eigen API key openbaar online hebben staan (op bijvoorbeeld GitHub, of live op een webadres). Iedereen die de broncode inspecteert, zou dan namelijk jouw API key kunnen gebruiken. Dit geldt ook voor belangrijke database credentials of andere afgeschermd tokens. Dit probleem lossen we op via environment variables.

Environment variables zijn variabelen die we declareren in een .env bestand. Omdat we dit bestand moet pushen naar GitHub (en dus altijd even handmatig moeten toevoegen aan de .gitignore!) blijven onze tokens geheim. Waarschijnlijk denkt je nu: maar als we met anderen samenwerken aan een project, hebben zij de geheime token toch ook nodig? En dat klopt ook. Daarom maak je ook altijd een .env.dist of .env.example file aan die je wel meeplukt naar GitHub. Daarin zet je de namen van de variabelen, maar zonder waarden. Het is dan voor de volgende developer duidelijk welke tokens ze nodig hebben om het project te laten werken. In de README.md benoem je altijd hoe men aan deze waarden kan komen (even een collega of baas vragen) - of ons geval: dat ze zelf een API key moet maken bij OpenWeather.

Oké, you get the pic. Maar hoe doen we dit?

- Maak in de root map op de desbetreffende hogepte de .gitignore en package.json een .env én een .env.dist file aan.
- Voeg het woord .env toe aan het .gitignore bestand.
- Open de .env.dist en zet daar REACT_APP_API_KEY=. We daartoe om environment variabelen in hoofdletters te schrijven. Bovendien moeten de namen altijd met REACT_APP beginnen, omdat het framework van Create React App ze anders niet kan vinden.
- Kopieer de naam van deze variabele en plak 'n in .env. Plak de waarde van jouw API key er achter (`green spaces & quotes`):

```
REACT_APP_API_KEY=d85e6256cb39361faf2b9d7d84acc4eb33f4
```

• Run het commando npx run build in de terminal.

Waarna? Het is alweer een tijds gelezen - maar weet je nog hoe React werkt onder de motorkap? Onze browser heeft totaal geen boodschap aan al die React code.

Na iedere aanpassing in het .env bestand zal je de development server opnieuw moeten starten

Door middel van Webpack worden al onze losse JavaScript, HTML en CSS bestanden samengevoegd tot één grote statische bundel code. Dit noemen we de build. Een nieuw .env-bestand is pas onderdeel van het project wanneer er minstens één build is gedraaid, omdat deze buiten de src-map staat. Wanneer je daarna een aanpassing maakt in het .env bestand, kan dit niet tijdens runtime (als jouw project draait) worden uitgelezen. Je zult hiervoor de development server eerst moeten stoppen (CTRL + C) en opnieuw moeten starten (npx start).

We zijn maar aan deze variabele te gebruiken! Deze is beschikbaar op het process.env object.

- Open App.js en verwijder `process.env` declaratie;
- Vervang de plek in de url waar de apikey gebruikt werd door `process.env.REACT_APP_API_KEY`:

```
try {
  const result = await axios.get(`https://api.openweathermap.org/data/2.5/weather?q=${location},nl&appid=${process.env.REACT_APP_API_KEY}&lang=nl`);
  setWeatherData(result.data);
} catch (error) {
  console.error(error);
  setError(true);
}
```

• Doe dit nu ook in de TodayTab.js en de ForecastTab.js:

Goed bezig! Wanneer je dit doet voor jouw eigen projecten, voeg je in de README.md een begeleidende tekst toe waarin je de gebruiker instructeert een eigen .env bestand aan te maken en deze te vullen met de variabel-namen zoals beschreven in env.dist. Uiteraard benoem je ook het maken van een build, en waar de gebruiker aan deze variabelen kan komen.

Let op: voor jouw eigen project doe je altijd jouw eigen API keys in de installeerhandleiding, zodat de examiner niet zelf een key hoef aan te maken.

9. Context

9.1 Inleiding

Als op een React applicatie bouwt geef je continu data door via properties. En hoe meer kleine, herbruikbare componentjes je bouwt, hoe groter de kans dat je sommige stukjes data meerdere lagen diep moet doorgeven. Dit kan er soms voor zorgen dat alle tussenliggende componenten een "doorgedrukt" worden voor die één button die - 5 niveau's diep - moet weten wat de gebruikersnaam is van de ingelogde gebruiker. Sterker nog: in sommige situaties moeten zelfs blauw en oranje componenten die elkaar niet eens kennen, dezelfde informatie delen.

- Een applicatie heet een provider, is, waarbij alle zijn componenten zijn ver de hoogte gescheiden tot;
- een applicatie waarin een enkelvoudige component wordt teruggegeven die de gebruiker controleert (pagina's en producten visual);
- een applicatie heet waarbij ingelogde gebruikers allemaal dezelfde context kunnen bekijken. Als pagina-componenten moeten weten of de gebruiker ingelogd en geautoriseerd is om de content te bekijken;
- een light- en dark-mode interface heet, waarbij alle componenten met één oog op de kleur moeten wennen van styling;

Dit zou ervoor kunnen zorgen dat een soortgelijke situatie ontstaat:

```
function App() {
  return (
    <>
      <Header theme={theme} />
      <Main theme={theme} />
      <Sidebar theme={theme} />
      <Footer theme={theme} />
    </>
  );
}

function Header({ theme }) {
  return (
    <>
      <NavBar theme={theme} />
      <LoginForm theme={theme} />
    </>
  );
}
```

Het probleem? We zijn data aan het doorgeven aan componenten die het zelf niet per se nodig hebben, maar deze properties alleen ontvangen omdat één of meerdere van hun child-componenten er gebruik van willen maken. Dit noem je **property drilling**. Het zou dus handiger zijn als de <NavBar> en <LoginForm> deze data zelf konden aanspreken.

React Context lost dit probleem voor ons op. Context is als het ware een overkoepelend stukje state waar ieder component in de applicatie bij zou kunnen. Wanneer een component de informatie uit de context nodig heeft - een consumer - kan het direct aangesproken worden, omdat een top-level component deze data in de context aanbiedt (provider).

Context versimpelt

Laten we contact illustreren met een (overdreven) simpelst voorbeeld. Stel dat we context gebruiken voor een tweedelige website. Allereerst moeten we de context natuurlijk creëren. Dit doen we met de `createContext` functie van React. Net als bij het aanmaken van state, geef je deze functie altijd een initiele waarde mee. In dit voorbeeld is dat null:

```
const LanguageContext = React.createContext(null);
```

We geven de Context altijd een naam die beschrijft wat voor soort data het bevat. Let op: het is belangrijk hiervoor een hoofdletter te gebruiken. Nu we Context hebben, hebben we ook toegang tot het Provider-component dat zich standaard op de Context bevindt. Deze wikkelen we altijd om het meest top-level component in de applicatie (vaak <App />):

```
// index.js
<LanguageContext.Provider value={{ activeLanguage: 'nederlands' }}>
  <App />
</LanguageContext.Provider>
```

Alles wat we meegeven aan het value-attribut, wordt beschikbaar in de Context. In dit geval geven we een object met daarin de key activeLanguage mee, die de string 'nederlands' bevat. Als er nu een component binnen onze applicatie gebruik wil maken van deze data, hoeven we alleen maar de useContext-hook van React te gebruiken om die toegang te verschaffen:

```
import LanguageContext from './LanguageContext';
function RandomComponent() {
  const language = React.useContext(LanguageContext);

  return (
    <p>
      De huidige taal is {language}
    </p>
  );
}
```

Side note: totale hooks werden geïntroduceerd in React, werkten we altijd met een .Provider en een .Consumer component. Het gebruik van useContext() in plaats van het .Consumer component, heeft het gebruik van context een stuk makkelijker gemaakt.

En klaar is Kees!

Ben je er klaar voor om Context in de praktijk te brengen? In de volgende paragraaf ga je Context gebruiken om de temperatuur in de Weather App te kunnen wisselen tussen de eenheid Celsius en Fahrenheit.

9.2 Het Context.Provider component

Het .Provider-component zorgt ervoor dat onze data beschikbaar is in de Context. In het voorbeeld in de vorige paragraaf hebben we alleen een waarde in de Context geplaatst. In de praktijk is het echter vaak zo dat je niet alleen een waarde beschikbaar wil stellen, maar ook een functie die die waarde aan kunnen passen vanuit een component. Kijk maar naar onze Weather App: het `MetricSlider`-component moet de geselecteerde temperatuur-eenheid, die opgetekend staat in de Context, kunnen passen als de gebruiker een andere eenheid kiest. En op basis van die selectie moeten de andere componenten hun temperatuur in Kelvin omrekenen naar Celsius of Fahrenheit. Het is daarom gebruikelijk om de Context op te zetten als een stukje state.

• Maak een nieuw geraamde context aan in de src-dir;

• Maak in die map een bestand genaamd `TempProvider.js` (lat op: met hoofdletter)

• In dit bestand beginnen we met het creeren van een `TempContext` met de `createContext` functie. Deze krijgt een initiale waarde van null

• Zet het woordje `export` voor `const`. We hebben onze context steeds immers ergens anders nodig!

```
import React, { createContext } from 'react';
export const TempContext = createContext(null);
```

Super, het goed is! Nu willen we een `TempContext.Provider` gaan opzetten. Herinner je je het voorbeeld in de vorige paragraaf nog?

```
<languageContext.Provider value={{ activeLanguage: 'nederlands' }}>
<App />
</languageContext.Provider>
```

zoals je ziet willen we dus een `.Provider` om het `<App>`-component heen wikkelen. Maar we willen ook dat de data die via de `.Provider` beschikbaar wordt gemaakt, middels een stukje state wordt bijgehouden. Hier hebben we een eigen-gemaakte component voor nodig dat onze `TempContext.Provider` returned, én een stukje state mee geeft. Een soort jusje voor onze Provider. Laten we beginnen met het toevoegen van een eigen component in `TempProvider.js`:

```
import React, { createContext } from 'react';
export const TempContext = createContext(null);

function TempContextProvider() {
  // hier komt straks de state waarin we de context-data plaatsen
  return (
    <TempContext.Provider value={{}}>
      // hier komt het component waar we onze eigen provider omheen wikkelen
    </TempContext.Provider>
  )
}

export default TempContextProvider;
```

Eigenlijk doen we nets meer, dan een component om de "basis" heen wikkelen. Om er nu voor te zorgen dat we straks ons `TempContextProvider`-component om `<App>` heen kunnen wikkelen, geven we de `property children` mee. Dit doen we als volgt:

```
function TempContextProvider({ children }) {
  return (
    <TempContext.Provider value={{}}>
      {children}
    </TempContext.Provider>
  )
}
```

Dit property kun je niet herroepen naar buiten: het is namelijk standaard functionaliteit van React. Je zou dit bijvoorbeeld ook met een button kunnen doen als je die later nog om tekst heen zou willen wikkelen.

```
function Button({ children }) {
  return (
    <button type="button">
      {children}
    </button>
  )
}
```

Oké, terug naar ons component. Laten we onze `<TempContextProvider>` eerst om het `<App>`-component wikkelen:

- Open `index.js` en importeer daar ons `<TempContextProvider>`-component
- Wikkel hem, binnen de `<React.StrictMode>`-tags, om het `<App>`-component heen

Heb je dat op de volgende manier gedaan?

```
import React from 'react';
import TempContextProvider from './context/TempContext';

ReactDOM.render(
  <React.StrictMode>
    <TempContextProvider>
      <App />
    </TempContextProvider>
  </React.StrictMode>,
  document.getElementById('root'),
);
```

Kijk eens aan! Je hebt zojuist de basis voor de `TempContext` opgezet. In de volgende paragraaf leer je hoe je data en functies in de Context plaatst.

Bekijk [dit](#) hoe de applicatie er nu uit ziet.

9.3 Data in de context plaatsen

Op dit moment is het `value`-attribuut op de `TempContext.Provider` nog ongeldig leeg... Wat voor data moeten we eigenlijk bijhouden voor de Context in onze Weather App? Nou, laten we beginnen met de geselcteerde temperatuur-eenheid!

- Maken in `TempContextProvider`-component een state staal met de key `selectedMetric`, waarbij de initiale waarde de string '`'celcius'`' is.

We willen nu straks voor zorgen dat ons `<TempContextProvider>`-component gebruik kan maken van de `setSelectedMetric`-functie om de `selectedMetric` te kunnen veranderen. Om de `selectedMetric` te kunnen veranderen, moeten we ons echter bewust zijn van de huidige `selectedMetric`. Als het Celsius is moet het naar Fahrenheit, en als het Fahrenheit is moet het naar Celsius. Een belangrijke regel met Context is dat je altijd wil proberen om **alle logica in het Provider component te houden**. We willen liever niet dat Consumer-componenten zelf moeten gaan berekenen welke nieuwe eenheid er in de state geplaatst moet worden. Dit zou namelijk ook een hoop dubbele code opleveren. Daarom schrijven we een dergelijke functie, die gebruik maakt van onze state-setter functie, direct uit in het Provider component:

```
function TempContextProvider({ children }) {
  const [selectedMetric, toggleSelectedMetric] = useState('celcius');

  function toggleTemp() {
    if (selectedMetric === 'celcius') {
      toggleSelectedMetric('fahrenheit');
    } else {
      toggleSelectedMetric('celcius');
    }
  }

  return (
    //...
  );
}
```

Super. Nu hoef je onder straks, zonder enige extra logica, alleen maar de `toggleTemp`-functie aan te roepen. Maar dit zullen we straks pas gaan implementeren. Ons Provider component is nog niet af! We willen er ook voor zorgen dat we, op basis van de geselcteerde eenheid, ook de juiste omreken-functie in de Context plaatsen. We gebruiken nu overal de `kelvinToCelsius` functie, maar we zullen dus ook een helper-functie nodig hebben om Kelvin naar Fahrenheit om te rekenen!

- Maken een nieuw bestandje in de `helpers`-map en noem deze `kelvinToFahrenheit.js`.
- Schrijf en exporteer hierin de `kelvin` als parameter verwacht en een fahrenheit string teruggeft. De formule kan je gemakkelijk op internet vinden. Verwachte uitkomsten: 304 geeft 88° F en 289 geeft 61° F.

Top, ziet dat er zo uit?

```
function kelvinToFahrenheit(kelvin) {
  return `${Math.round((kelvin - 273.15) * 1.8) + 32}° F`;
}

export default kelvinToFahrenheit;
```

- Importeer nu beide helper-functies in `TempContextProvider.js`.

We gaan ons `value`-attribuut vullen met een object, zodat we zowel onze `toggleTemp` als de helper-functies kunnen doorgeven:

```
return (
  <TempContext.Provider
    value={{
      toggleTemp,
      kelvinToMetric: selectedMetric === 'celcius' ? kelvinToCelsius : kelvinToFahrenheit,
    }}
  >
    {children}
  </TempContext.Provider>
)
```

Zoals je ziet geven we, op basis van de waarde van de `selectedMetric`, respectievelijk de functie `kelvinToCelsius` of `kelvinToFahrenheit` mee. Ook nu hoeven de componenten die deze functie gaan gebruiken, zich niet bezig te houden met de logica. Ze hoeven slechts de `kelvinToMetric` functie te gebruiken zonder zich bewust te zijn van de huidig geselcteerde eenheid.

Side note: als de object key `selectedMetric` naam heeft als de variabele die je er in stopt (`toggleTemp: toggleTemp`) mag je er ook échtegaal wegeën: (`toggleTemp`). De working blijft hetzelfde!

Bekijk [dit](#) het einderesultaat van ons Provider.

9.4 Context consumeren

We hebben nu Context aangemaakt, een Provider component opzettig die data in de Context zet en deze om ons `<App>`-component gewikkeld. Het is tijd voor het Consumer-gedoe! Er is nu immers nog geen enkel component die onze Context gebruikt. Laten we daar verandering in brengen.

Open `ForecastTab.js`. We hebben twee dingen nodig:

- Onde `TempContext`
 - De `useContext` hook van React
- Importeer beiden bovenaan het bestand.

Het enige wat we nu hoeven te doen, is onze functie die we onder de key `kelvinToMetric` in de Context hebben gezet, weer te destructuren uit de Context:

```
import React, { useEffect, useState, useContext } from 'react';
import { TempContext } from '.././context/TempContextProvider';
const { kelvinToMetric } = useContext(TempContext);
```

• Vervang nu de `kelvinToCelsius` functie door onze `kelvinToMetric` functie uit de Context.

```
<section className="forecast-weather">
  <span>
    {kelvinToMetric(forecast.temp.day)}
  </span>
</section>
```

Dat was simpel! Ons `<ForecastTab>`-component is nu "gebonneerd" op de Context en zal automatisch update als er iets in de Context verandert.

- Open nu `WeatherDetail.js` en importeer daar ook de `TempContext` en de `useContext` hook.
- Haal ook hier de `kelvinToMetric` functie uit de context en verwang `kelvinToCelsius`.
- In `App.js` wordt ook nog een `kelvinToCelsius` functie gebruikt. Vervang deze ook door onze Context-omreken-functie.
- Sup! Nu gaan we onze slider ook nog abonneren op de Context. Open `MetricSlider.js`. Zoals je ziet, is onze slider eigenlijk een checkbox die door CSS het uiterlijk van een slider heeft gekregen. Om onze slider functioneel te maken, moeten we er eerst een `controlled component` van maken.
- Mak de checkbox een controlled component door een state aan te maken die `toggled` tussen `true` en `false`. De intiale waarde is `true`.
- Importeer de `TempContext` en de `useContext` hook en substitueer daar de `toggleTemp` functie uit.
- Zorg ervoor dat wanneer de state van de checkbox verandert, de `toggleTemp` functie wordt aangeroepen (fp: gebruik een Life Cycle method).

Als het goed is, heb je dit op de volgende manier gedaan:

```
const MetricSlider = () => {
  const [checked, toggleChecked] = useState(true);
  const [ toggleTemp ] = useContext(TempContext);

  useEffect(() => {
    toggleTemp();
  }, [checked]);

  return (
    <input
      type="checkbox"
      className="switch"
      id="metric-system"
      checked={checked}
      onChange={() => toggleChecked(!checked)}
    />
  );
}
```

Side note: gezien we slechts de `toggleTemp` functie aanroepen in de `useEffect` functie, hadden we dat ook zo kunnen schrijven:

```
useEffect(toggleTemp, [checked]);
```

Je bent klaar! Je hebt in dit hoofdstuk geleerd hoe je context maakt, probeert en consumiert. En hoewel Context op het eerste gezicht vaak nog een wat lastiger te begrijpen concept is, heb je ook kunnen zien dat je het in drie simpele stappen zelf kan implementeren!

Bekijk het eindresultaat van de Weather App [hier](#).

10. Authenticatie

10.1 JSON Web Tokens

Waarschijnlijk heb je al een API gezien maar dan een manier om data uit te wisselen met een externe server. De server heeft een aantal endpoints beschikbaar waar wij netwerk-requests naartoe kunnen maken. Veel van deze endpoints zullen beveiligd zijn. We willen immers niet dat gebruiker a een blogpost kan plaatsen onder de naam van gebruiker b, dat iemand anders jouw persoonlijke accountgegevens kan aanpassen of dat gevoelige informatie voor leden beschikbaar is.

We zullen dus bij elke request naar een beveigde endpoint moeten laten zien wie wij zijn, zodat de server kan checken of wij geautoriséerd zijn om deze informatie aan te spreken. Eén van de manieren om dat te doen, is door het gebruik van **JSON Web Tokens**, afgekort tot JWT. Een JWT bevat gegevens over onze identiteit, die versleuteld zijn tot een Base64 string, zoals bijvoorbeeld deze:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIi0iIxMjM0NTY30DkwIiwibmFtZSI6IkpvA4gRG91IiwiWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5cwh
```

Wanneer wij intussen in de frontend (door onze gebruikersnaam en wachtwoord via een POST-request naar de server te sturen) zal de server ons zó'n versleutelde token toestaan. Vanaf dat moment zullen we deze token goed moeten bewaren, omdat we hem met alle volgende netwerk-requests moeten meesturen als bewijs van authenticatie.

Dit JWT blijft geldig tot de opgegeven geldigheidsduur - bepaald door de server - verstrekken is. Er wordt dus bij elke request opnieuw gevalideerd of deze token nog geldig is. Maar hoe wordt zo'n JWT eigenlijk opgebouwd?

Well, het is het juist bovenstaand voorbeeld al opgegeven dat de token bestaat uit drie verschillende delen, opgesplitst door een `.`. Elk van de drie delen bevat andere informatie.

Encoded

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIi0iIxMjM0NTY30DkwIiwibmFtZSI6IkpvA4gRG91IiwiWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5cwh
```

Decoded

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYOUT: DATA

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
) □ secret base64 encoded
```

- Het eerste deel bevat de **token-header**, waarin onder andere vermeld wordt welk versleutelingsalgorithmie er is gebruikt. Voorbeelden hiervan zijn HMAC, HS256 of RSA.
- Het middelste deel bevat de **payload** waarin informatie over de gebruiker, de uitgever van de token en vaak de geldigheidsduur verscholen zit. Een voorbeeld van die (onversleutelde) payload kan zijn:

```
{
  "sub": "1234567890",
  "exp": 1239843374692,
  "username": "pieterertje"
}
```

- Het derde deel bevat de geheime handtekening, ofwel **signature**. In tegenstelling tot de vorige twee delen - die gemakkelijk te decoderen zijn - is de signature niet te decoderen voor vreemden omdat hier een private key voor gebruikt is. Op deze manier is het goed te valideren dat de gebruiker echt is wie hij of zij beweert dat hij is.

Omdat de informatie uit de eerste twee delen van de token zo gemakkelijk te decoderen is, wil je hier nooit gevoelige informatie (zoals wachtwoorden) in opnemen.

Decoderen

In de frontend willen we soms gebruikmaken van de informatie die in een JWT verscholen zit, zoals de geldigheidsduur of de gebruikers-id van de gebruiker voor wie deze token bedoeld was. In dat geval zullen de token moeten decoderen. Gelukkig heeft iemand dit wel al lang uitgevonden en daar een handige npm package voor gemaakt: [jwt-decode](#). Om het gebruik van te maken, installeren we deze package in ons project met het commando:

```
npm install jwt-decode
```

Vervolgens importeren we deze package in het bestand waar we een token willen decoderen, en roepen we de `jwt-decode` functie aan. Het resultaat is een object met daarin de gedecodeerde informatie:

```
import jwt_decode from 'jwt-decode';
function App() {
  const token = eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJdWIi0iIxMjM0NTY30DkwIiwibmFtZSI6IkpvA4gRG91IiwiWF0IjoxNTE2MjM5MDIyfQ.Sf1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c;
  const decoded = jwt_decode(token);
  return (

```

```
}); //...
}
```

Requests maken met token

Wanneer je requests gaat doen naar beveiligde endpoints, zul je dus continu jouw JWT mee moeten sturen. Deze token voeg je toe aan de request header, onder de key `Authorization`. We gaan er in onderstaande voorbeeld uit gemak even vanuit dat onze token de string `xxxxx-yyyyy-zzzzz` is.

Get request

Bij axios GET-requests is het tweede argument altijd het object met de request header, indien je ervoor kiest deze mee te sturen. Hierin plaatsen we onze token, vergelijkbaar door de woordje "Bearer". Je noemt het principe van token-authenticatie ook wel bearer authentication, vandaar!

```
axios.get('https://...', {
  headers: {
    "Content-Type": "application/json",
    "Authorization": "Bearer xxxxx-yyyyy-zzzzz",
  },
});
```

Post request

Bij axios POST-requests is het derde argument altijd het object met de request header, indien je ervoor kiest deze mee te sturen. Hierin plaatsen we onze token. Vaak is het dan ook nuttig om het type van de door jou meegestuurde data te specificeren.

```
axios.post('https://...', {
  comment: 'Super lekker recept!',
  headers: {
    "Content-Type": "application/json",
    "Authorization": "Bearer xxxxx-yyyyy-zzzzz",
  },
});
```

10.2 Local Storage

Zodra de gebruiker ingelogd is, zullen een aantal componenten de gebruikersnaam of het e-mailadres van de ingelogde gebruiker willen weergeven. Dsze willen we daarom direct in de Context plaatsen. Wellicht is het ook handig om het ID van de gebruiker in de Context voorhanden te hebben, indien we die nodig hebben om andere gebruikersinformatie op te vragen. Vertrouwelijke informatie zoals wachtwoorden, of de meer gedetailleerde informatie die een gebruiker op zijn profielpagina zal vinden, plaatsen we hier niet in.

Dit zal alsnog voorspoedig verlopen, totdat de gebruiker de pagina refresh. Once applicatie wordt opnieuw gestart, de Context wordt opnieuw gemount en al onze data is verdwenen. Dit is natuurlijk een bekend probleem: we kunnen immers geen data opslaan in de frontend. Hoe lossen we dit op?

Gelukkig is er voor dit soort situaties wel een mogelijkheid om piepjonge stukjes data op te slaan in de browser van de gebruiker. Dit noemen we **Web Storage**. We zullen hier nooit gebruikersinformatie in opslaan, maar ons belangrijkste stukje data, de JWT token, kunnen we daar moet in bewaren!

Web Storage bevat u **Sessie Storage** of **Local Storage**. Session storage blijft bestaan zodra de gebruiker zijn browser-tab open laat staan. Local Storage blijft zolang bewaard als we willen en is in het geval van authenticatie dus het meest geschikt. Open EdHub maar eens in jouw browser. Je kunt jouw eigen Local Storage bekijken als je de console open en op het tabje "Application" klikt. Vouw links de Local Storage open en -hef! Daar staat ook een Bearer token!

Application		Filter
	Key	Value
Storage	showAppsMenu	true
	sidebarCollapsed	true
	token	Bearer eyJhbGciOiJIUzI1NilsInR5cCl6IkpxVCJ9eyJleHAiOjE2MzM3MTgzNDEsImF1dGh...
Local Storage	showNotifications	true
	https://edhub.novi.nl	
	Session Storage	
	IndexedDB	
	Web SQL	
	Cookies	
	Trust Tokens	

Informatie in de Local Storage plaatsen is verrassend eenvoudig. Hier gebruiken we de methodes van het ingebouwde `localStorage`-object voor.

Data in de Local Storage plaatsen

Wanneer we data in de Local Storage zetten, mag je zelf bedenken onder welke key je dat wilt doen. In ons geval is de key `token` het meest logisch, maar bananen had ook genomen. We gebruiken hier de `setItem` methode voor, waarin de eerste parameter de naam van de key is, en de tweede parameter de waarde.

```
localStorage.setItem('token', 'xxxxx-yyyyy-zzzzz');
```

Data uitlezen

Wanneer we data willen uitlezen, gebruiken we de `getItem`-methode. We moeten hierbij wel opgeven welke key we willen uitlezen. Het resultaat staan we dan op in een variabele.

```
const token = localStorage.getItem('token');
```

Data verwijderen

Je kunt de Local Storage in één keer helemaal leegpogen met de `clear`-methode, of één specifieke key verwijderen door middel van de `removeItem`-methode:

```
localStorage.removeItem('token');
// of
localStorage.clear();
```

Het is niet de bedoeling om de token zowel in de Context als in de Local Storage te plaatsen, dit is een **anti-pattern** (een aanpak die we als bad practise beschouwen).

10.3 Authenticatie implementeren

In deze paragraaf leggen we je uit hoe je Context, JWT's en Local Storage laat samenwerken om de belangrijkste authenticatie-functionaliteiten in jouw applicatie te implementeren. Dit betreft inloggen, uitloggen en ingelogd blijven wanneer de pagina vernieuwd wordt (**persist on refresh**).

Inloggen

Wanneer de gebruiker zijn gegevens invoert in een invoerveld, worden deze middels een POST-request verstuurd naar de backend. Indien deze gebruikersnaam- en wachtwoord combinatie klopt, stuurt de backend een reactie terug. Wat er precies in deze reactie zit, is afhankelijk van de implementatie van deze backend:

1. De backend stuurt alleen de JWT terug;
2. De backend stuurt zowel de JWT en de belangrijkste gebruikersinformatie, zoals gebruikersnaam, profielfoto en e-mailadres terug.

De laatste optie is voor de huidige implementatie van React het handigste, omdat we daarna geen extra requests meer hoeven te doen naar de backend. Als frontend ontwikkelaar heb je dit echter niet altijd voor het zeggen. Vervolgens geven we deze JWT en eventuele andere gebruikersinformatie door aan de `login`-functie van de Context. Deze zorgt ervoor dat de JWT in de Local Storage terecht komt en de gebruikersgegevens in de Context-state.

Uitloggen

Op het moment dat een gebruiker het proces van uitloggen in gang zet, wordt de `logout`-functie van de Context aangeroepen. Die zorgt ervoor dat de gebruikersgegevens verwijderd worden uit de Context-state en de Local Storage wordt ontslaan van de JWT.

Persist on refresh

Wanneer de applicatie opnieuw geladen wordt, zal eerst de state geinitialiseerd worden met een lege user-key. Op dat moment weet de Context dus niet of zijn gebruikersdata afwezig is omdat er écht niemand is, of omdat we data zijn kwijtgeraakt door een refresh. Vervolgens worden alle componenten in de applicatie gerenderd. Tijdens het mounting-effect zal de Context vervolgens altijd even moeten checken of er nog een geldige token in de Local Storage staat. Als dat zo is, zal de Context daarmee een GET-request moeten doen naar de server om gebruikersdata op te vragen. Wanneer dit binnen en in de Context-state is geplaatst, kunnen de andere componenten hier ook gebruik van maken.

Als je nadinkt over de Life Cycle van React en de vorige paragraaf nog een keer leest, heb je dan al een vermoeden wat er misgaat in dit scenario?

De volgende applicatie, met al haar componenten en private routes, zal al gerenderd worden vóórdat er gebruikersdata in de context geplaatst is. Tijdens deze render-base is de Context-state nog leeg en zullen de private routes ten onrechte aannemen dat de gebruiker niet gauthoriseerd is. Pas dáarna zal het mounting-effect getriggert worden, de Local Storage worden uitgelezen, de data worden opgehaald en ten slotte verwerkt.

We willen dus dat alle children van ons Context-component pas gerenderd worden nadat onze check voltooid is.

Dit kunnen we gemakkelijk oplossen door onze state van een soort "status" te voorzien. Initieel staat deze nog op 'pending' (wanneer de applicatie wordt opgestart of gerefreshed). Wanneer dit zo is, zorgen we in de return-statement van het custom Provider-component dat we <p>loading...</p> renderen, in plaats van de gehele applicatie (<App />). Daarna kan ons mounting-effect getriggert worden, die gaat checken of er een token in de Local Storage staat:

- Indien deze niet gevonden wordt, zal de gebruiker null blijven, maar wordt de status wel op 'done' gezet. Omdat de state is gewijzigd wordt de conditie opnieuw gevalideerd. Omdat de status nu op 'done' staat, zal de applicatie gerenderd zonder ingelogde gebruiker.
- Indien deze wel gevonden wordt, kan er nieuwe gebruikersinformatie opgeladen worden. Deze wordt vervolgens in de state gezet, tezamen met 'done' in de status-key. Omdat de state is gewijzigd, wordt de conditie opnieuw gevalideerd. De applicatie wordt gerenderd, wetende dat er een gebruiker ingelogd is.

```
function AuthContextProvider({ children }) {
  const [authState, setAuthState] = useState({
    user: null,
    status: 'pending',
  });

  useEffect(() => {
    // check of er nog een token in Local Storage staat
    // ZO JA: haal dan de nieuwe data op en zet deze in de state:
    setAuthState({
      user: {
        username: 'Piet',
        email: 'pieter@gmail.com',
        id: 23,
      },
      status: 'done',
    });
    // ZO NEE:
    setAuthState({
      user: null,
      status: 'done',
    });
  }, []);

  const data = {
    ...authState,
    login: login,
    logout: logout,
  };

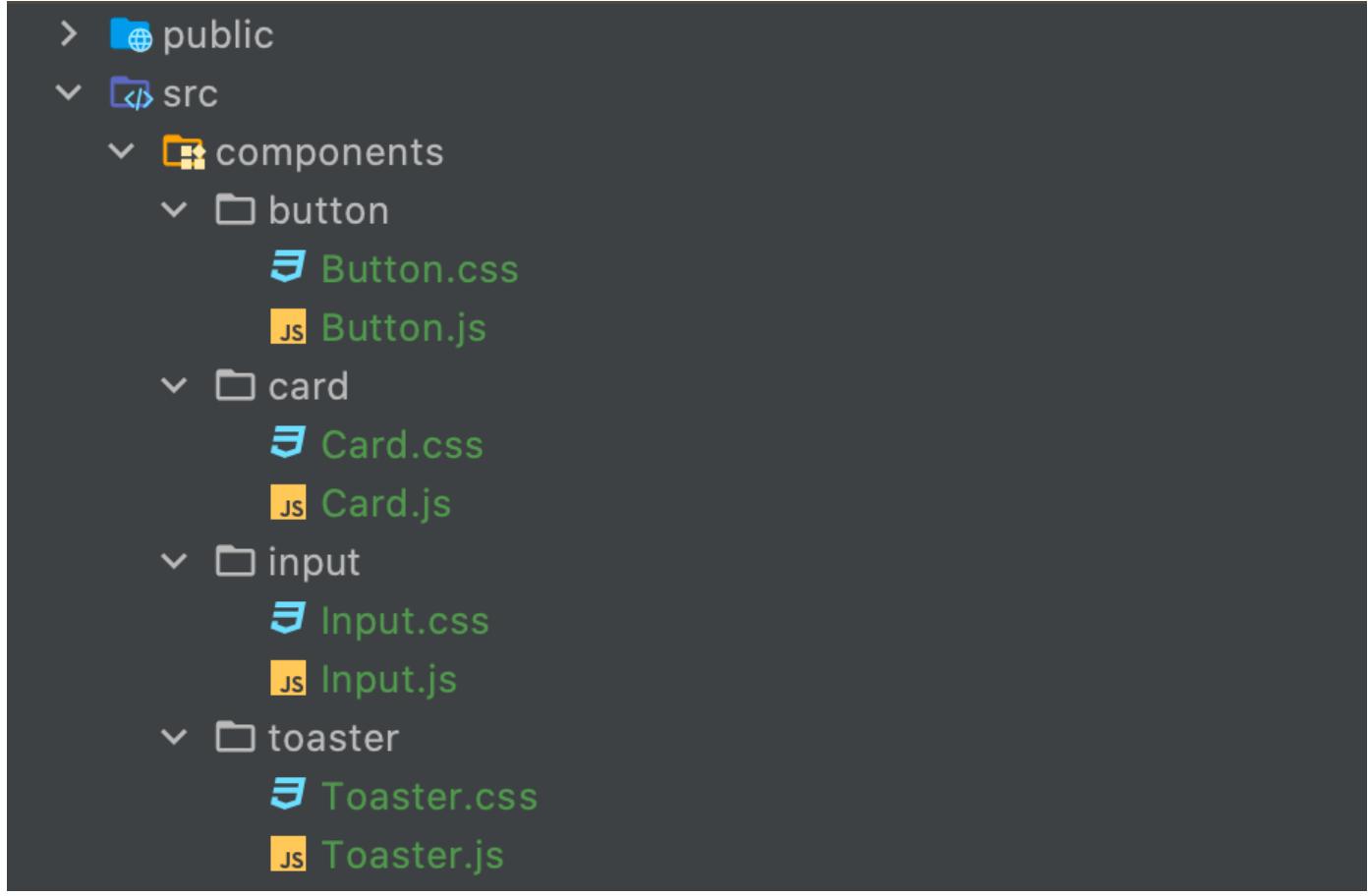
  return (
    <AuthContext.Provider value={data}>
      {authState.status === 'pending'
        ? <p>Loading...</p>
        : children
      }
    </AuthContext.Provider>
  );
}

export default AuthContextProvider;
```

11. Bonus: styling in een grote applicatie

11.1 CSS Modules

Tot nu toe heb je waarschijnlijk altijd vanuit één groot CSS-bestand gewerkt. En hoevel dat makkelijker was om te oefenen, is dit niet hoe je het normaal gesproken aanpakt. Het is een best practice om per React component een bijbehorend CSS-bestand aan te maken, zodat het altijd duidelijk is waar de styling bij hoort:



Dit wordt aangemoedigd in React, omdat we een bibliotheek van herbruikbare componenten bouwen. Componenten (en hun styling) worden éénmalig geschreven en daarna hergebruikt op verschillende plekken. Maar er is één grote valkuil. Wanneer we in 'button.js' ons bijbehorende CSS bestand importeren -

```
import './Button.css';
```

- wakt die indruk dat de CSS alleen invloed kan hebben op onze 'button.js'. Dat is echter niet zo, en dat heeft te maken met hoe React ontwerpen is. Laten we die motoriek nog eens openbreken. React is een prachtige uitvinding voor ons developers, maar de browser is niet in staat om React code te lezen en daar een webpagina van te maken. De browser "sprekt" alleen plain HTML, plain CSS en plain JavaScript. Het is met de hulp van Webpack dat al onze losse React componenten, JavaScript-en CSS-bestanden worden samengevoegd tot één grote statische bundel code die de browser wel begrijpt. Dit betekent dat wanneer je een class .error declareert in 'Button.css', je deze class ook gewoon kunt gebruiken in 'Toaster.js', 'Input.js' of 'Card.js'. Al die losse CSS wordt, uiteindelijk, immers samengevoegd één groot CSS bestand. En dat zorgt natuurlijk snel voor problemen als je bent vergeten dat er een .error class in 'Button.css' staat, en je schrijft een nieuwe .error class in 'Card.css'. En bij je maar afvragen waarom de CSS van juur error-card maar niet doet wat je verwacht...

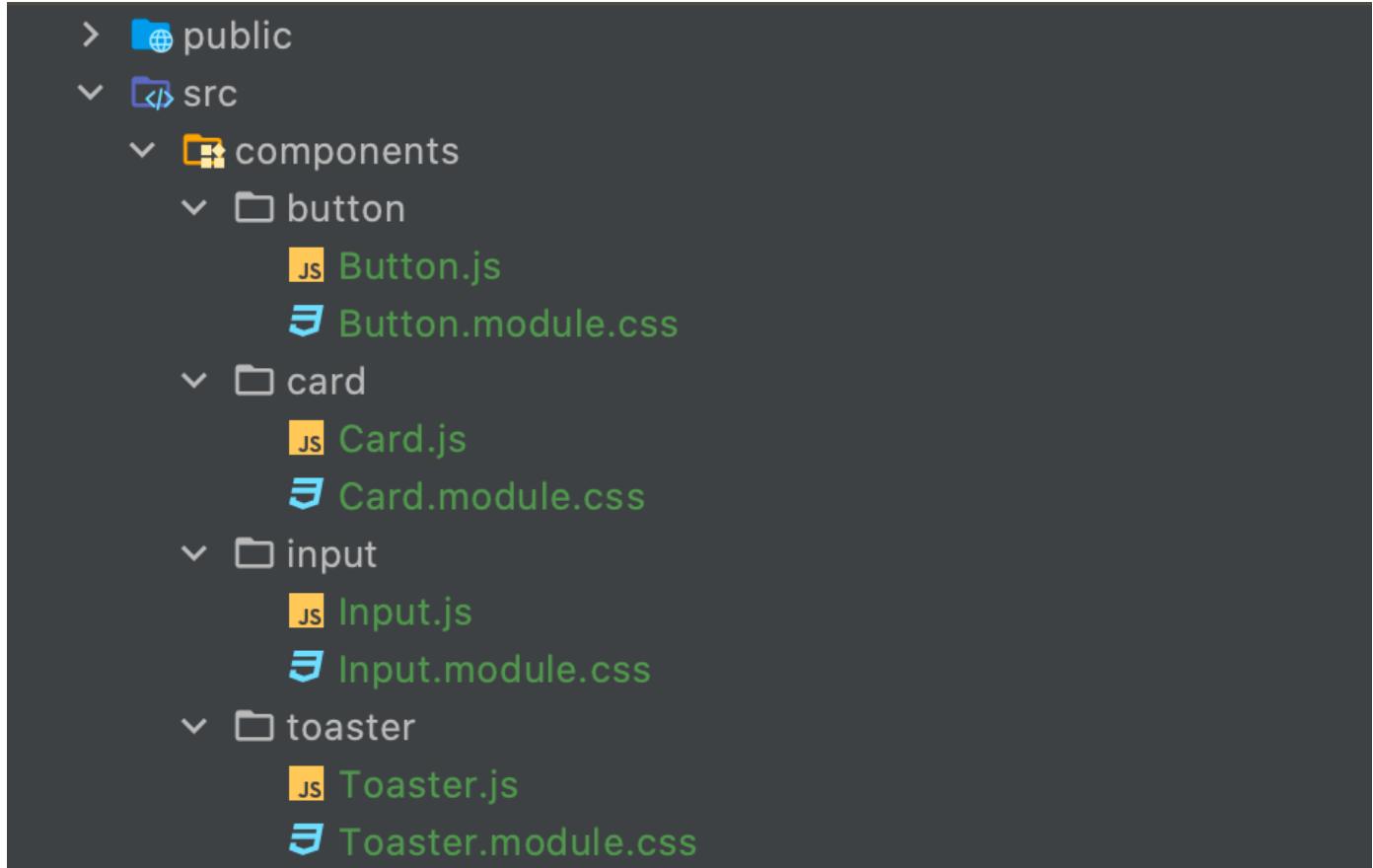
Je zou je blijkbaar zorgen gaan maken. Maar gelukkig hebben we hier een oplossing voor: CSS modules. Die zorgen ervoor dat classes niet meer global, maar alleen lokaal worden toegepast, omdat ze bij het compileren (verzamelen) worden aangevuld met een soort unieke hash, waardoor de class-naam uniek blijft. Implementatie bestaat uit drie simpele stappen:

1. Hernoem je bestanden
2. Importeer de CSS-bestanden onder de naam styles
3. Speek CSS-classes in het component aan onder styles.naam-van-de-class

Laten we dat iets praktischer uitleggen:

1. Hernoem je bestanden

Het enige wat je hoeft te doen, is de CSS bestanden te hernoemen naar 'Button.module.css' in plaats van 'Button.css':



2. Importeer de CSS-bestanden onder de naam styles

Vervolgens pas je de import statement aan door de module.css te importeren onder de naam styles (of banan, mag ook)

```
// GENONE CSS
import './Button.css';
// CSS MODULES
import styles from './Button.module.css';
```

Spreek CSS-lassen aan met styles.naam-van-de-class

Bet dat we een class error en een class default-button hebben met daarin wat styling. Met gewone CSS hebben we die alrijf met className="error" op onze elementen geplaatst. Met CSS Modules, zit ze nu beschikbaar als property op het styles object dat we gelimposeerd hebben: className={styles.error}. Wanneer een class -takens bevat, zoals default-button (dit bestaat niet in een normale object notatie), moeten we er blokkenhaken en quotes () omheen zetten. Dat is een regel in JavaScript. Voor alle namen zonder smoege, gebruik je de normale object notatie.

```
import styles from './Button.module.css';
function Button() {
  return (
    <>
      <button type="button" className={styles['default-button']}>
        Versturen
      </button>
      <p className={styles.error}>
        Error!
      </p>
    </>
  )
}
```

Wat er nu precies veranderd is, kun je goed zien met Google Dev Tools. In het onderstaand voorbeeld zie je de vergelijking van het eindresultaat van dezelfde CSS-class (.default-button) respectievelijk met- en zonder een CSS module.

```

<html lang="en">
  > <head>...</head>
  > <body cz-shortcut-listen="true">
    >> <div id="root">
      >>> <div>
        ><button type="button" class="Button_default-button__2fA_u">Versturen</button>
        ><button class="default-button">Versturen</button>
      ></div>
    ></div>
    <script src="/static/js/bundle.js"></script>
    <script src="/static/js/0.chunk.js"></script>
    <script src="/static/js/main.chunk.js"></script>
  </body>
</html>

```

CSS Module

Gewone CSS

11.2 BEM

Styling begint altijd leuk, maar CSS-bestanden kunnen vrij snel de spulletjes uitleppen. Bovendien weet je na een tijdje van gekheid ook niet meer wat voor classNaam je nu weer voor dat extra element moet verzinnen. Gelukkig ben je hierin niet de enige. Er zijn meerdere manieren die je kunt gebruiken om je CSS overzichtelijk te houden, en **BEM** is daar één van. BEM staat voor **Block-Element-Modifier**. Het is een naming-convention standaard voor CSS klassen. En in normaal Nederlands: een setje aan regels waarop je je kunt houden bij het bedenken van namen voor je CSS-klassen.

Geen rocket science dus!

Een BEM-class bestaat uit maximaal drie delen:

1. Block: het basisde parent element van het component
2. Element: een specifiek type zijn dat waarop enkele modulaire children aanwezig
3. Modifier: wanneer een block of een element meerdere variabelen heeft (bijvoorbeeld een info en error button)

Wanneer deze drie delen gebruikt worden in een naam, zou dat er zo uitzien:

[block]_[element]--[modifier]

Zet er leuk uit, maar dit is nog een beetje abstract. Laten we naar wat voorbeelden gaan kijken!

Een Component zonder Elementen of Modifiers

Het meest simpele component dat verder geen child elementen bevat, krijgt alleen een basis-class-naam:

```
<button className="btn"></button>

// styles.css
.btn {}
```

Een Component met een Modifier

Een component kan meerdere variaties kennen. In dit geval wordt de variatie geïmplementeerd met een modifier-class **bovenop** de basis-class. Deze herken je aan de dubbele --:

```
<!-- DOE DIT -->
<button className="btn btn--error"></button>

// styles.css
.btn {
  display: inline-block;
  color: blue;
}
.btn--error {
  color: red;
}
```

Let op: gebruik een modifier-class nooit in z'n eerst! Het is niet de bedoeling dat de basis-class **vervangen** wordt, hij wordt slechts **uitgebreid** met een modifier-class.

```
<!-- DOE DIT NIET -->
<button className="btn--error"></button>

// styles.css
.btn--error {
  display: inline-block;
  color: red;
}
```

Een Component met Elementen

Zodra je een component iets meer functionaliteit geeft, zal deze direct child-elementen bevatten. Zoals bijvoorbeeld een afbeelding of een titel in een <fig>. Omdat BEM wil voorkomen dat je in een soort nesting half rechtt komt (de a in de 11 in de u1 in de ... etc.) dwingt het af dat je ieder element een eigen class-naam geeft - zolang dit element ook daadwerkelijk van styling moet worden voorzien.

```
<!-- DOE DIT -->
<figure className="photo">
  
  <figcaption className="photo__caption">
    Een foto van mij in 2001
  </figcaption>
</figure>

// styles.css
.photo { }
.photo__img { }
.photo__caption { }

<!-- DOE DIT NIET -->
<figure className="photo">
  
  <figcaption>Een foto van mij in 2001</figcaption>
</figure>
```

// styles.css

.photo { }

.photo img { }

.photo figcaption { }

Als het component child elementen bevat die meerdere niveaus diep zijn, probeer dan niet elk niveau in de klassennaam weer te geven. BEM is niet bedoeld om de diepte van de structuur uit te beelden. Een BEM-klassennaam die een child element vertegenwoordigt, mag alleen een block en één element-naam bevatten. Bekijk het onderstaande voorbeeld eens. `photo__caption__quote` is geen juist gebruik van BEM, maar `photo__quote` wel.

```
<!-- DOE DIT -->
<figure className="photo">
  
  <figcaption className="photo__caption">
    <blockquote className="photo__quote">
      Een foto van mij in 2001
    </blockquote>
  </figcaption>
</figure>

// styles.css
.photo { }
.photo__img { }
.photo__caption { }
.photo__quote { }
```

```
<-- DOE DIT NIET -->
<figure className="photo">
  
  <figcaption className="photo__caption">
    <blockquote className="photo__caption_quote">
      Een foto van mij in 2001
    </blockquote>
  </figcaption>
</figure>

// styles.css
.photo {
  .photo__img { }
  .photo__caption { }
  .photo__caption_quote { }
}
```

Een Element met Modifier

In sommige gevallen wil je wellicht individuele items binnen een component laten variëren. In dat geval kun je een Modifier op het element plaatsen, zoals het voorbeeld hieronder. In de praktijk is het echter vaak zo dat je een modifier toevoegt aan het volledige component, omdat er meerdere onderdelen mee veranderen.

```
<figure className="photo">
  
  <figcaption className="photo__caption photo__caption--large">
    Een foto van mij in 2001
  </figcaption>
</figure>

// styles.css
.photo__img--framed { }
.photo__caption--large { }
```

Als je merkt dat je elementen van één component consequent samen op dezelfde manier aanpassen, overweeg dan om de Modifier aan de basis van de component toe te voegen. Zo kun je styling toepassen door de bestaande elementen als children aan te spreken. Dit verhoogt wel de specificiteit, maar het maakt het wijzigen van het component veel eenvoudiger:

```
<-- DOE DIT -->
<figure className="photo photo--highlighted">
  
  <figcaption className="photo__caption photo__caption--highlighted">
    Een foto van mij in 2001
  </figcaption>
</figure>

// styles.css
.photo--highlighted .photo__img { }
.photo--highlighted .photo__caption { }

<-- DOE DIT NIET -->
<figure className="photo">
  
  <figcaption className="photo__caption photo__caption--highlighted">
    Een foto van mij in 2001
  </figcaption>
</figure>

// styles.css
.photo__img--highlighted { }
.photo__caption--highlighted { }
```

Meerdere woorden samenvoegen

BEM namen maken opzettelijk gebruik van dubbele tekens (- en -), zodat je meerdere woorden aan elkaar nog steeds kunt scheiden met één streepje. We willen class-namen zo leesbaar mogelijk houden, dus voeg meerdere woorden niet samen:

```
<-- DOE DIT -->
<div className="important-section important-section--fast-read">
  <div className="important-section__other-element"></div>
</div>

// styles.css
.important-section {
  .important-section--fast-read {
    .important-section__some-element { }

<-- DOE DIT NIET -->
</div> De classnamen zijn veel moeilijker te lezen!
<div className="importantsection importantsection--fastread">
  <div className="importantsection__someelement"></div>
</div>

// styles.css
.importantsection {
  .importantsection--fastread {
    .importantsection__someelement { }
```

11.3 SASS en SCSS

Syntactically Awesome Style Sheets, ook bekort tot SASS, is volgens de makers simpelweg CSS met superkrachten. Het is als het ware een verlengstuk van CSS waarmee je concepten als variabelen, nesting, en herbruikbare stukjes styling kunt gebruiken. Bovendien helpt het jouw styling overzichtelijk te houden. SASS wordt uiteindelijk door Webpack weer terugvertaald naar CSS, dus het is vooral een hulpsysteem voor jou als ontwikkelaar.

SASS heeft twee syntax-vormen:

- SCSS ("Sassy CSS") als separator van CSS. Dit betekent dat ieder valide stukje CSS ook geldig is als SCSS.
- SASS is de originele syntaxis, ooit ontwikkeld voor developers die behoefted belangrijker vonden dan het behouden van gelijkenissen met CSS. Dus in plaats van het gebruik van accolades en puntkomma's, maak je gebruik van indentatie om declaratie aan te duiden.

Als je met bovenstaande uitleg niet veel wijzer bent geworden, bekijk dan onderstaand voorbeeld eens:

```
/* SCSS */
$blue: #3bbffce;
$margin: 16px;

.content-navigation {
  border-color: $blue;
  color: darken($blue, 9%);
}

.border {
  padding: $margin / 2;
  margin: $margin / 2;
  border-color: $blue;
}

/* SASS */
$blue: #3bbffce
$margin: 16px

.content-navigation
  border-color: $blue
  color: darken($blue, 9%)

.border
  padding: $margin / 2
  margin: $margin / 2
  border-color: $blue
```

In alle verdere voorbeelden zullen we de meer bekende SCSS syntax gebruiken. Om SASS (en dus SCSS) te kunnen gebruiken in je project, run je het volgende commando in je terminal:

```
npm install sass --save-dev
```

Omdat SASS ons alleen helpt tijdens het ontwikkelen, is het geen gewone dependency, maar een dev dependency. Als je al CSS bestanden in je project had staan, is het belangrijk om die bestandsnamen van .css te hernoemen naar .scss.

Side note: tot 26 oktober 2020 was het gebruikelijk om node-sass (ofwel libSass) in plaats van sass te gebruiken in create-react-app projecten. De library sass geeft toegang tot de nieuwste (experimentele) features van SASS, maar compiled niet zo snel als node-sass. Het team van SASS heeft echter toch besloten om niet meer verder te ontwikkelen aan node-sass, en raden aan bij toekomstige projecten gebruik te gaan maken van sass. Lees [hier meer over de onderliggende redenen](#).

Oké, maar je vraagt je nu waarschijnlijk af: wat kun je er precies mee?

Variabelen

CSS variabelen kende je natuurlijk al - maar ook SCSS kent variabelen, en ze zijn iets makkelijker te gebruiken. En hoevel variabelen erg simpel zijn, zijn ze toch ontzettend krachtig. Het is gebruikelijk om één (of meerdere) aparte .scss bestanden aan te maken waarin je een lijst veel-gebruikte waarden in variabelen declareert. Denk hierbij aan:

- Alle kleurcodes die in je project gebruikt worden
- Standarde font-groottes
- Standarde padding- en margin-aanpassingen

Je declareert een variabele simpelweg door er een \$ voor te zetten. Bovendien kun je variabelen ook weer in andere variabelen gebruiken. Kijk maar eens naar dit voorbeeld:

```
// variables.scss
$base-color: #c6538c;
$border-dark: rgba($base-color, 0.88);
```

Als we één van deze variabelen nu willen gebruiken in een ander SCSS-bestand, gebruiken we hier een @import statement voor:

```
// HomePage.scss
@import '../variables.scss';
.alert {
  border: 1px solid $border-dark;
}
```

Nesting

Hoeveel we volgens BEM juist zo min mogelijk willen nesten, maakt SCSS het gelukkig wel een stuk makkelijker in het geval dat we dat toch willen doen. Vergelijk de volgende voorbeelden eens met elkaar. Het eerste voorbeeld is "gewone" CSS, het tweede voorbeeld gebruikt SCSS:

```
/* CSS
nav ul {
  margin: 0;
  padding: 0;
  list-style: none;
}

nav li {
  display: inline-block;
}

nav a {
  display: block;
  padding: 6px 12px;
  text-decoration: none;
}

/*SCSS
nav {
  ul {
    margin: 0;
    padding: 0;
    list-style: none;
  }

  li {
    display: inline-block;
  }

  a {
    display: block;
    padding: 6px 12px;
    text-decoration: none;
  }
}
```

En dit geldt ook voor pseudo-selectors, waarbij je simpelweg het & teken kunt gebruiken als de selector moet worden toegepast op het huidige element. Maar ook op het nesten van property-selectors die bij elkaar horen:

```
/*CSS*
.enlarge {
  font-size: 14px;
  transition-property: font-size;
  transition-duration: 4s;
  transition-delay: 2s;
}

.enlarge:hover {
  font-size: 36px;
}

/*SCSS*
.enlarge {
  font-size: 14px;
  transition:
    property font-size;
    duration: 4s;
    delay: 2s;
}

&:hover {
  font-size: 36px;
}
```

Mixins

Hoeveel React er juist voor zorgt dat we zo min mogelijk dubbele code (en CSS) hoeven te schrijven, zal je aan sommige dingen simpelweg niet ontkomen. Zoals bijvoorbeeld het schrijven van een media-query en het gebruik van breakpoints. Terwijl je graag alle breakpoints uit je hoofd leert, is dit een hele fijne manier om het schrijven van media-queries makkelijker te maken.

Een mixin kenmerkt zich door de directive @mixin, gevolgd door een zelfbedachte naam. Daarbij kun je de directive @content gebruiken, net als dat we in React (children) gebruiken om aan te geven dat we het component ergens overheen kunnen wikkelen. Kijk bijvoorbeeld eens naar het volgende voorbeeld:

```
@mixin for-tablet-and-up {
  @media (min-width: 768px) {
    @content;
  }
}

@mixin for-desktop-and-up {
  @media (min-width: 1200px) {
    @content;
  }
}

.container {
  font-size: 12px;
  @include for-tablet-and-up {
    font-size: 14px;
  }
}
```

12. Bonus: vroegere Class Components

12.1 Bonus: React Hooks vs class components

Programmeertalen, libraries en frameworks kenmerken zich door het feit dat ze continu verbeterd en doorontwikkeld worden. Een taal zoals JavaScript of een library als React is nooit "af". De functionaliteit en schrijfleerpjes van React zag er heel anders uit toen het net gelanceerd werd, in vergelijking tot nu. En dat in slechts een paar jaar tijd! Je begrijpt het al: we kunnen dus niet in het verleden blijven hangen.

Tot 2019 zag het React-landschap er iets ingewikkelder uit dan nu. We gebruikten naast function components (die ken je immiddels) ook class components. Een class component was de enige manier om gebruik te kunnen maken van state. Dit zorgde er in de praktijk voor dat frontend developers veel tijd kwijt waren aan refactoring function components omzetten naar class components wanneer ze erachter kwamen dat een component toch state nodig had. Niet zo handig dus.

Gelukkig kwam hier in februari 2019 verandering in: React lanceerde **React Hooks**. Dit maakt het gebruik van class components overbodig en vereenvoudigt het gebruik van state en andere belangrijke React functionaliteiten. Hooks werden immiddels wereldwijd toegepast, maar logischerwijs zal er nog een hoop documentatie op internet te vinden zijn die gebaseerd is op class components. Daarom is het belangrijk om class components te herkennen en te weten hoe je voorbeelden op internet kunt ontschrijven naar een function component met hooks.

Later lezen teruggaan naar een paar jaar terug. Destijds was de regel dat wanneer je geen state nodig had in een component, je daar altijd een function component voor gebruikte. Deze waren namelijk een stuk minder "zwaar" dan class components. Neem bijvoorbeeld een EmailField-component zonder state:

```
function EmailField({ disabled }) {
  return (
    <input
      type="email"
      disabled={disabled}
    />
  );
}

export default EmailField;
```

Dit zie je bekend uit: Als we hier met React Hooks nu state aan toe willen voegen met een externe event-handler-functie, hoeven we slechts een paar regels code toe te voegen:

```
function EmailField({ disabled }) {
  const [emailValue, setEmailValue] = React.useState('');
  function handleInput(e) {
    setEmailValue(e.target.value);
  }

  return (
    <input
      type="email"
      disabled={disabled}
      value={emailValue}
      onChange={handleInput}
    />
  );
}
```

Easy! Maar stel dat we nu een aantal jaar terug in de tijd gaan, naar de periode voordat React Hooks bestonden. Als we ons simpele function component nu van state en een externe event-handler-functie willen voorzien, moet het hele component op de schop! Voordat we state kunnen toevoegen, zullen we ons function component eerst moeten omschrijven naar een class component. Dat zie je zo uit:

```
class EmailField extends React.Component {
  constructor(props) {
```

```

    } super(props);
}

render() {
  return (
    <input
      disabled={this.props.disabled}
      type="email"
    />
  );
}

export default EmailField;

```

Een class component krijgt altijd een constructor() en een render() functie. De constructor zorgt ervoor dat we properties door kunnen geven aan onze parent-class: React.Component, die ervoor zorgt dat we via this.props bij onze properties kunnen. Wanneer je gebruik maakt van classes, krijg je automatisch ook te maken met het this keyword waarop alle variabelen en methoden van de class te vinden zijn. Alles wat gereturneerd wordt uit de render-functie, wordt gerendered (weergegeven) op de pagina.

Maar we zijn er nog niet! Nu kunnen we pas beginnen met het toevoegen van state en externe event-handler-functie:

- In de constructor creëren we de state variabele;
- Deze state variabele kunnen we binnen de render-functie aanspreken via this.state.emailValue;
- We declareren een event-handler-methode op ons class component. Om ervoor te zorgen dat deze ook netjes via this beschikbaar is, moeten we deze methode binden in de constructor door middel van this.handleInput = this.handleInput.bind(this);. Wil je hier meer over weten? Hoe binden precies werkt en waarom je dit nodig hebt in class components, wordt uitgebreid uitgelegd in [de artikel](#).
- Ten slotte kunnen we onze methode meegeven via de onChange-propertie.

```

class EmailField extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      emailValue: '',
    };
    this.handleInput = this.handleInput.bind(this);
  }

  handleInput(e) {
    this.setState({
      emailValue: e.target.value,
    });
  }

  render() {
    return (
      <input
        type="email"
        disabled={this.props.disabled}
        value={this.state.emailValue}
        onChange={this.handleInput}
      />
    );
  }
}

export default EmailField;

```

Wanneer je deze uitwerkingen naast elkaar zet, zie je pas hoe groot het verschil in regels code is:

Class component

```

class EmailField extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      emailValue: '',
    };

    this.handleInput = this.handleInput.bind(this);
  }

  handleInput(e) {
    this.setState({
      emailValue: e.target.value,
    });
  }

  render() {
    return (
      <input
        type="email"
        disabled={this.props.disabled}
        value={this.state.emailValue}
        onChange={this.handleInput}
      />
    );
  }
}

export default EmailField;

```

Functie component met hooks

```

function EmailField({ disabled }) {
  const [emailValue, setEmailValue] = React.useState('');

  function handleInput(e) {
    setEmailValue(e.target.value);
  }

  return (
    <input
      type="email"
      disabled={disabled}
      value={emailValue}
      onChange={handleInput}
    />
  );
}

export default EmailField;

```

React hooks hebben ons leven dus een stuk makkelijker gemaakt! Dat betekent echter niet dat de React community in één keer overstapt op deze nieuwe variant. Het betekent ook niet dat alle oude tutorials- of artikelen herschreven zullen worden. Het is daarom aan jou om class components te herkennen in ouder materiaal en dit zelf toe te passen in de nieuwe schrijfwijze. Het is namelijk absoluut niet de bedoeling om class components en hooks door elkaar te gebruiken.

Welke variant kies jij?

Kijk op: tijdens de React Converte 2018 hebben de makers van React een hele [introductie sessie](#) gegeven over de nieuwe features van React. Vanaf minuut 11:30 verteld Dan Abramov in 20 minuten op een zeer begrijpelijke manier welke problemen React tot nu toe ervaren heeft met class components en waarom React Hooks zo'n groot verschil zal gaan maken.

12.2 Bonusopdracht: van class component naar hooks

In veel documentatie zul je voorbeelden tegenkomen met class components. Daarom is het goed om ook deze code te kunnen lezen, begrijpen, en omschrijven naar een functiecomponent met hooks. Je kunt de opdracht downloaden of downloaden naar jouw eigen computer via [deze GitHub repository](#). De uitwerkingen staan op de branch uitwerkingen.

Opdrachtbeschrijving

In onderstaand voorbeeld vindt je een class component met daarin een lichtknopje. De staat van het lichtknopje (aan of uit) wordt bijgehouden in de state. De tekst op de button wordt ook aangepast op basis van de state:

```

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isLightOn: true
    };
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState({isLightOn: !this.state.isLightOn});
  }
  render() {
    return (
      <button
        type="button"
        onClick={this.handleClick}
        >{this.state.isLightOn ? 'AAN' : 'UIT'}
      </button>
    );
  }
}
export default App;

```

1. Kopieer de code van bovenstaand voorbeeld en bekijk de code regel-voor-regel. Schrijf commentaar bij elke regel code, waarin je in jouw eigen woorden beschrijft wat deze regel doet. Wanneer je een specifieke regel code niet begrijpt, kun je het ook proberen te Googleen.

2. Bekijk het hele document met jouw commentaar. Heb je voor ieder hader wat er gebeurt? Zo ja, dan kun je beginnen met het maken van een functie-component.

Applicatie starten

Als je het project gedownload hebt naar jouw lokale machine, installeer je eerst de node_modules door het volgende commando in de terminal te runnen:

```
npm install
```

Wanneer dit klaar is, kun je de applicatie starten met behulp van:

```
npm start
```

of gebruik de WebStorm knop (npm start). Open <http://localhost:3000> op de pagina in de browser te bekijken. Begin met het maken van wijzigingen in src/App.js; elke keer als je een bestand opslaat, zullen de wijzigingen te zien zijn op de webpagina.

12.3 Life Cycles in class components

Class components bestaan al veel langer dan hooks. Daarom behandelen we ook kort hoe de life cycles werken met class components, mocht je deze voorbeelden ooit tegenkomen in documentatie of oudere code. Je hoeft dus niet precies te weten hoe ze werken, zolang je begrijpt waar de methodes voor dienen. In plaats van één useEffect-hook, hadden class components een aparte methode voor iedere life cycle, namelijk:

Life Cycle	Status	Trigger functie
Initialisatie	State initialiseren	<code>constructor()</code>
	Vóór het component toegevoegd wordt	<code>componentWillMount() *</code>
Mounting	Wanneer het component wordt gerenderd (toegevoegd) aan de DOM	<code>render()</code>
	Als het component eenmaal gerenderd is	<code>componentDidMount()</code>
	Vóór het geupdate in het component toegevoegd wordt	<code>componentWillUpdate() *</code>
Updating	Wanneer het component gerenderd wordt aan de DOM met nieuwe waardes	<code>render()</code>
	Nadat de properties van het component veranderd zijn en het component eenmaal is aangepast in de DOM	<code>componentDidUpdate()</code>
Unmounting	Vlak voor het component verwijderd wordt uit de DOM	<code>componentWillUnmount()</code>

*Dit betekent dat React het gebruik van deze functies niet meer ondersteund omdat ze onstabiel gedrag kunnen veroorzaken.

Laten we dit eens in een component bekijken. De volgorde waarin je de lifecycle methods toekoest aan een class component heeft geen invloed op de volgorde waarin ze worden uitgevoerd. Ze kunnen dus allemaal door elkaar staan, maar worden pas getriggert op het moment dat het component zich in die life cycle bevindt:

```

class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
    console.log('State is aangemaakt!');
  }

  // UPDATE CYCLE
  componentDidUpdate(prevProps, prevState) {
    if (prevState.count != this.state.count) {
      console.log('State is veranderd, dus ik ga updaten!');
    }
  }

  // UNMOUNTING CYCLE
  componentWillUnmount() {
    console.log('Ik ga zo verwijderd worden uit de DOM');
  }

  // MOUNTING CYCLE
  componentDidMount() {
    console.log('Ik ben nu in de DOM gezet! Check.');
  }

  render() {
    return (
      <h1>Het aantal is {this.state.count}</h1>
    );
  }
}

```

Best cool om te bedenken dat we al deze functies tegenwoordig vervangen hebben door één useEffect-hook!

Je zult deze methoden ongetwijfeld nog tegenkomen in documentatie of oudere React code. Mocht je meer willen lezen over de life cycle methoden van class components, is dit een [interessant artikel](#).

13. Cursusevaluatie - React

Je hebt zojuist de cursus React afgerond. We zijn benieuwd wat je ervan vindt! Om ervoor te zorgen dat wij continu kunnen blijven verbeteren, vragen we jou om deze cursus (de content, lessen en docent) te evalueren. Dit kan via:

[Evaluatie - Cursus React](#)

Het invullen van de vragenlijst kost je 1 à 2 minuten en is anoniem. Dan krijg je!