# Samenvatting react

[Samenvatting react 1](#_Toc140496967)

[Voorbeeld 3](#_Toc140496968)

[Zie ook 3](#_Toc140496969)

[Te gebruiken bij: 3](#_Toc140496970)

[Eigenschappen: 3](#_Toc140496971)

[React de bassis 3](#_Toc140496972)

[Wat is React? 3](#_Toc140496973)

[Kenmerken van React 4](#_Toc140496974)

[Voordelen van React 4](#_Toc140496975)

[JavaScript taal en de JavaScript Xml (JSX javascript extension) extensie onder de loep. 5](#_Toc140496976)

[JSX 5](#_Toc140496977)

[Waarom JSX? 5](#_Toc140496978)

[JSX-elementen 5](#_Toc140496979)

[JSX-elementen en veiligheid 8](#_Toc140496980)

[**Injection safe** 8](#_Toc140496981)

[andere frameworks 8](#_Toc140496982)

[AngularJS 8](#_Toc140496983)

[VueJS 9](#_Toc140496984)

[JQuery 9](#_Toc140496985)

[cursus react (nieuw) 9](#_Toc140496986)

[Inleiding 9](#_Toc140496987)

[React 10](#_Toc140496988)

[Component library 10](#_Toc140496989)

[JavaScript XML(**JSX**) 11](#_Toc140496990)

[De virtuele DOM 12](#_Toc140496991)

[Wat gebeurt er in de virtuele DOM 14](#_Toc140496992)

[Een eigen project opzetten 14](#_Toc140496993)

[npm install vs npm start 14](#_Toc140496994)

[De basis 16](#_Toc140496995)

[HTML en CSS in React 16](#_Toc140496996)

[oefening 16](#_Toc140496997)

[app.js 16](#_Toc140496998)

[Fragments(<> </>) 18](#_Toc140496999)

[Eigen afbeeldingen en iconen gebruiken 19](#_Toc140497000)

[SVG’s 19](#_Toc140497001)

[Event listeners 20](#_Toc140497002)

[Verdieping: imports en exports 21](#_Toc140497003)

[**default exports** 21](#_Toc140497004)

[**named exports en import** 22](#_Toc140497005)

[Componenten en properties 23](#_Toc140497006)

[Twee cruciale verschillen 23](#_Toc140497007)

[Herbruikbare react componenten 23](#_Toc140497008)

[Properties 24](#_Toc140497009)

[props 24](#_Toc140497010)

[Keys 24](#_Toc140497011)

[Speciale props doorgeven: children 25](#_Toc140497012)

[Interactieve componenten en formulieren 26](#_Toc140497013)

[State 26](#_Toc140497014)

[hooks 28](#_Toc140497015)

[Condities 29](#_Toc140497016)

[Conditioneel stylen 30](#_Toc140497017)

[Conditionele properties 31](#_Toc140497018)

[Props vs. Callback props 32](#_Toc140497019)

[callbacks 32](#_Toc140497020)

[Controlled components 33](#_Toc140497021)

[uncontrolled components 34](#_Toc140497022)

[controlled components 34](#_Toc140497023)

[Maar hoe werkt dit? 34](#_Toc140497024)

[Meerdere inputs met één onChange-handler 35](#_Toc140497025)

[onChange 35](#_Toc140497026)

[Een enkel input veld 36](#_Toc140497027)

[Meerdere input velden 37](#_Toc140497028)

[Spread 38](#_Toc140497029)

[Selectbox en/of Textarea 38](#_Toc140497030)

[Checkboxes en radio buttons 39](#_Toc140497031)

[Bonus: React Hook Form 40](#_Toc140497032)

[librarys 40](#_Toc140497033)

[Installatie 40](#_Toc140497034)

[Invoervelden registeren 41](#_Toc140497035)

[Registratie 41](#_Toc140497036)

[Useform 41](#_Toc140497037)

[Implimentatie van React Hook Form 46](#_Toc140497038)

[Submitten 46](#_Toc140497039)

[Validatie regels 48](#_Toc140497040)

[Foutmeldingen toevoegen 49](#_Toc140497041)

[Foutmeldingen weergeven in de UI 50](#_Toc140497042)

[Conditionele velden 51](#_Toc140497043)

[Formulier opties 52](#_Toc140497044)

[mode 52](#_Toc140497045)

[Prefilling 53](#_Toc140497046)

[Van elementen naar componenten 53](#_Toc140497047)

[Routing 57](#_Toc140497048)

[Routes definiëren 59](#_Toc140497049)

[routes 59](#_Toc140497050)

[Links 60](#_Toc140497051)

[NavLinks 60](#_Toc140497052)

[link 60](#_Toc140497053)

[Nalink 61](#_Toc140497054)

[**Hoe werkt de callback van NaLink?** 62](#_Toc140497055)

[Doorlinken 63](#_Toc140497056)

[Dynamische routes 64](#_Toc140497057)

[hoe weet de pagina nou welk product de gebruiker op dat moment bekijkt...? 65](#_Toc140497058)

[dynamische parameters 65](#_Toc140497059)

[userParams-hook 65](#_Toc140497060)

[Beveiligde routes 66](#_Toc140497061)

### Voorbeeld

### Zie ook

* ZIE: SAMENVATTING X

### Te gebruiken bij:

-

### Eigenschappen:

-

### React de bassis

#### Wat is React?

React is een JavaScript framework.

React is een manier om gebruikersinterfaces te bouwen. React maakt dit heel eenvoudig door elke pagina in “componenten” te knippen.



Een React-component is een stukje code dat een deel van de pagina vertegenwoordigt. Elke component is een JavaScript-functie die een stuk code retourneert.

Om een pagina te bouwen roepen we deze functies in een bepaalde volgorde aan, stellen het resultaat samen en laten het aan de gebruiker zien.

#### Kenmerken van React

* React gebruikt componenten om UI's te maken.   
  Componenten lijken op HTML-elementen met toegevoegde functionaliteiten om ze herbruikbaar te maken.
* In React bouw je ingekapselde componenten, vervolgens stel je deze componenten samen om complexe UI's te maken om ze herbruikbaar te maken.(DRY).
* je kunt nieuwe functies ontwikkelen in React zonder de bestaande code te herschrijven.

#### Voordelen van React

* Het renderen van componenten met React is zeer snel.
* React speelt goed met andere programmeertalen samen.
* React geeft de mogelijkheid om op elk gewenst moment code-elementen te hergebruiken, hetgeen een enorm tijdbesparend effect heeft.

Dit framework is ooit gestart om de grootste verschillen tussen de verschillende browsers voor programmeurs “onder de motorkap te verbergen” levert een standaard interface voor het afhandelen van grafische gebeurtenissen op het scherm.

het mogelijk om gegevens uit verschillende bronnen te combineren en samen te voegen in functies die HTML als uitvoer hebben.

uitgegroeid tot een van de meest gebruikte frameworks voor het ondersteunen van scherm gebruikersinteractie in HTML (op telefoons, desktop computers en tablets).

meest krachtige concepten binnen React: HTML en code zijn één.

#### JavaScript taal en de JavaScript Xml (JSX javascript extension) extensie onder de loep.

Naast het gebruik van JSX om HTML te genereren, gaan we ook een afbeelding creëren als Scalable Vector Graphics (SVG).

#### JSX

JSX is een extensie van JavaScript voor het gebruik van XML of HTML in JavaScript.

const element = <h1>Hello, world!</h1>;

Daarbij zijn de rood aangegeven elementen de HTML-elementen.

De React precompiler (Babel) zal de bovenstaande broncode vertalen naar JavaScript zodat de code te gebruiken is als element.

Buiten HTML zijn ook andere vormen van XML te gebruiken, zoals Scalable Vector Graphics (SVG): een manier om grafisch meer coherente elementen te maken.  
Denk hierbij aan grafieken, kaarten, animaties of een mooie lay-out

##### **Waarom JSX?**

* Bij veel applicaties is de presentatie zeer verweven met de logica van de applicatie. Het ligt dus voor de hand om dit op computertaal niveau te integreren.
* Een applicatie is makkelijker te maken als dingen die bij elkaar horen ook als een blok zijn te gebruiken. In de software engineering wereld heet dit Separation of Concerns (Dijkstra, 1982). Daarom is het makkelijker als deze blokken zowel de schermopmaak als de logica bevatten. Deze blokken heten in React componenten.
* Het voorkomt aanvallen op de applicatie door middel van injectie van code.

##### JSX-elementen

Om JSX dynamisch te maken is het mogelijk om stukjes JavaScript op te nemen in JSX.

zogenaamde expressies.

let username = “Jaap”;

const element = <h1>Hello, {username}!</h1>;

Het verrassende is dat JSX ook weer vertaald kan worden in expressies en dat deze expressies kunnen worden gebruikt in andere expressies.

Hetgeen weer leidt tot een groot aantal componenten, die kunnen worden gebruikt in een organisatie.

je zult dan ook zien dat in organisaties die ReactJs gebruiken de componenten vaak gedeeld worden tussen de verschillende applicaties.

Ook mag je meer geneste HTML-tags gebruiken.

Zie ook dat de h1-tags opgenomen zijn als kleine letters Dit is een voorwaarde: grote letters worden niet geaccepteerd door Babel.

Ook is het mogelijk om SVG te gebruiken in een JSX-statement:

import React from 'react';

function MyComponent() {

return (

<div>

<svg width="200" height="200">

<circle cx="100" cy="100" r="50" fill="red" />

</svg>

</div>

);

}

De gemaakte elementen kunnen vervolgens aan een render worden gegeven:

In de context van React verwijst "render" naar het proces waarbij een React-component wordt omgezet in een representatie in de werkelijke DOM (Document Object Model) die zichtbaar is in de webbrowser.

ReactDOM.render(

circle,

document.getElementById('root')

);

Nog makkelijker is het om de elementen te gebruiken in componenten.

Door React-elementen te gebruiken, kun je componenten op een declaratieve manier bouwen, waarbij je de hiërarchie en de relatie tussen componenten kunt definiëren met behulp van JSX-syntax. JSX biedt een beknopte en leesbare manier om React-elementen te maken en te gebruiken.

Hier is een voorbeeld van het gebruik van React-elementen in een component:  
In dit voorbeeld worden React-elementen zoals <div>, <h1>, en <p>

**met js**

jsx

Copy code

import React from 'react';

function MyComponent() {

const name = 'John Doe';

return (

<div>

<h1>Hello, {name}!</h1>

<p>This is a React component.</p>

</div>

);

}

In dit voorbeeld worden React-elementen zoals <div>, <h1>, en <p> gebruikt in JSX-syntax om de structuur en inhoud van de gebruikersinterface te beschrijven. JSX maakt het gemakkelijk om componenten te definiëren en te nesten, terwijl JavaScript-uitdrukkingen zoals {name} worden gebruikt om dynamische waarden in te voegen.

Door het gebruik van React-elementen in componenten kunnen de voordelen van React, zoals componenthergebruik, declaratieve syntax en efficiënte updates, ten volle worden benut. Het biedt een gestructureerde en intuïtieve manier om componenten te bouwen en de gebruikersinterface van een applicatie te beheren.

**Met een svg**

jsx

Copy code

import React from 'react';

function MyComponent() {

return (

<div>

<svg width="200" height="200">

<circle cx="100" cy="100" r="50" fill="red" />

</svg>

</div>

);

}

export default MyComponent;

In dit voorbeeld wordt het <svg>-element gebruikt als een React-element om een SVG-afbeelding weer te geven. Binnen het <svg>-element kun je andere SVG-elementen gebruiken, zoals <circle>, <rect>, <path>, enzovoort, om vormen en grafische elementen te tekenen.

Je kunt attributen, zoals width, height, cx, cy, r, en fill, rechtstreeks aan de React-elementen toewijzen om de eigenschappen van de SVG-elementen te configureren.

Op deze manier kun je SVG-elementen gemakkelijk opnemen in je React-componenten en ze gebruiken om complexe SVG-afbeeldingen te maken. Je kunt ook dynamische waarden gebruiken voor attributen door JavaScript-uitdrukkingen in te sluiten met behulp van {}.

##### JSX-elementen en veiligheid

###### **Injection safe**

Het kan gebeuren dat je data die door derden worden aangeleverd, wil tonen in een JSX-expressie.

Dat kan zonder dat je jezelf druk hoeft te maken dat die expressie geïnjecteerd kan worden met code van anderen.

geïnjecteerde code onschadelijk wordt gemaakt. Tijdens het vertalen door Babel

##### andere frameworks

###### AngularJS

AngularJS is een framework dat een soortgelijk doel nastreeft als ReactJS.

Het verschil zit in het gebruik van componenten. ReactJS heeft de mogelijkheid om componenten te maken en te hergebruiken. AngularJS heeft die mogelijkheid niet

<https://angular.io/>

###### VueJS

VueJS is een JavaScript-frontend framework dat is ontworpen om webontwikkeling te organiseren en te vereenvoudigen.

Het project richt zich op het toegankelijker maken van ideeën in webontwikkeling en UI-ontwikkeling (componenten, declaratieve gebruikersinterface, hot-reloading, debugging, enzovoorts).

Het probeert minder uitgesproken te zijn en dus gemakkelijker voor ontwikkelaars.

VueJS heeft een incrementeel toepasbare architectuur. De kernbibliotheek is gericht op declaratieve rendering en componentsamenstelling en kan worden ingesloten in bestaande pagina's.

Geavanceerde functies die vereist zijn voor complexe applicaties zoals routing, state management en build-tooling worden aangeboden via officieel onderhouden, ondersteunende bibliotheken en pakketten.

<https://vuejs.org/>

#### JQuery

JQuery kun je beschouwen als een minimaal framework voor het maken van dynamische websites.

Waar AngularJS en ReactJs faciliteiten hebben voor templates en componenten, heeft JQuery dit niet.

<https://jquery.com/>

### cursus react (nieuw)

Vergeet niet eerst een npm install te draaien zodat alle benodigde node\_modules worden gedownload. Daarna kun je de applicatie opstarten met het commando npm start.

#### Inleiding

Tot voor kort was het gebruikelijk om alle logica en automatisering te laten plaatsvinden in de back-end.

React is een front end library om gebruikersinterfaces mee te bouwen.

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 toepast in zo’n grote applicatie.

Bovendien zul je leren hoe de Life Cycles van React werken en hoe je deze gebruikt om jouw webapplicatie te voorzien van interactie en externe data.

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 principes 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.

JavaScript is in de afgelopen jaren een veel grotere rol gaan spelen in webapplicaties., veel processen en berekeningen uitvoeren in de browser, zonder met de server te hoeven communiceren.

Maar door de toegenomen complexiteit van applicaties is het ook steeds moeilijker geworden om JavaScript te onderhouden.

##### React

React gebruikt gestandaardiseerde Javascript structuren om orde in de chaos te scheppen en code makkelijker te onderhouden. Andere voordelen van deze library zijn:

* Het ontwikkelen van een applicatie is sneller en efficiënter omdat je herbruikbare User Interface componenten kan maken, zonder bestaande code te herschrijven;
* Een applicatie is schaalbaar en makkelijk uit te breiden met plugins;
* Code is netter en beter te onderhouden;
* Er is een enorme online community waardoor je makkelijk antwoorden kunt vinden op je vragen;
* Alle functionaliteiten van React zijn uitstekend gedocumenteerd;
* Het is, ten opzichte van andere frameworks zoals Angular, relatief makkelijk te leren.

##### Component library

React maakt gebruik van User Interface elementen die je kunt *hergebruiken*.

React componenten lijken op HTML-elementen, waarbij we specifieke functionaliteit toevoegen om ze herbruikbaar te maken.

We zien dat dezelfde categorie-button vaker terugkomt. Het enige verschil is de tekst op de button, het icoon, en de pagina waar hij naartoe linkt.

Met React kunnen we één generiek button component bouwen die telkens andere data accepteert, maar er hetzelfde uitziet en zich op dezelfde manier gedraagt.

* **Generieke componenten**: productblokken (geel), titels met subtitels (roze), categorie-knoppen (groen), menu-items (oranje)

Afbeelding met tekst, schermopname, meubels, ontwerp

Automatisch gegenereerde beschrijving

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.  
de buttons niet groen, maar roze moeten zijn? Geen probleem! We passen dit aan op één plek.

#### JavaScript XML(**JSX**)

React maakt gebruikt van **JSX**: JavaScript XML, het onze oude vertrouwde HTML in een JavaScript-jasje.( Met JSX kunnen we HTML elementen maken in een volledig JavaScript georiënteerde bibliotheek.

)

Om te begrijpen hoe JSX en React samenwerken, beginnen we bij het startpunt van elke webpagina: het index.html

<!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 één <div> in de body 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.

HTML-elementen door middel van JavaScript op de pagina zien te krijgen, dit doen we met JSX

Met JSX kunnen we HTML elementen maken in een volledig JavaScript georiënteerde bibliotheek.

Laten we bij het begin beginnen:

// We stoppen ons HTML-element in een JavaScript variabele

const pageTitle = <h1>JSX is cool!</h1>;

// We creêren 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 standaard in een element met de id root.

#### De virtuele DOM

Mogelijk heb al kennis gemaakt met Documnent Object Model

Maar hoe werkt React onder de motorkap? React maakt gebruik van een **virtuele DOM.**

Iedere keer als er een element aangepast moet worden op een webpagina, gebeurt dit via de DOM. Een node wordt toegevoegd, aangepast of verwijderd.

iedere keer als dit gebeurt, moet de browser opnieuw alle CSS berekenen, de layout bepalen en dit op het scherm ‘tekenen’.

React geprobeerd om de hoeveelheid DOM-veranderingen tot een minimum te beperken.

Afbeelding met tekst, schermopname, Lettertype, diagram

Automatisch gegenereerde beschrijving

Voor de "echte" DOM creëert React een een corresponderende "virtuele DOM": een lichtgewicht kopie van het origineel.

De virtuele DOM is licht omdat het eigenlijk een Javascript object is.

{

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',

},

],

},

],

}

;

##### Wat gebeurt er in de virtuele DOM

een class-attribuut willen toevoegen aan één van onze list items:

 Vervolgens vergelijkt React de originele virtuele DOM met de nieuwe virtuele DOM. Onthoud: omdat dit slechts JavaScript objecten zijn, kan dit razendsnel.

 Er ontstaat een *“diff”*: een 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.

#### Een eigen project opzetten

Om gebruik te maken van deze automatische React installatie heb je een versie van Node.js

Als jij 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'.

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>

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 eventjes duurt, komt omdat er *heel veel* node\_modules geïnstalleerd moeten worden. Een React project bestaat namelijk uit allemaal bouwstenen.

Zoals je weet, worden node\_modules nooit meegepusht 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, voer je daarom altijd eerst een globale installatie uit:

npm install

Hiermee haal je alle dependencies (node\_modules) van het project binnen.

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

npm run start

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

Afbeelding met tekst, schermopname, software, Besturingssysteem

Automatisch gegenereerde beschrijving

Wanneer je iets aanpast en opslaat worden de wijzigingen direct toegepast.

<https://vimeo.com/743053764/b5862eca09>

Afbeelding met tekst, schermopname, software, Besturingssysteem

Automatisch gegenereerde beschrijving

#### De basis

##### HTML en CSS in React

Inmiddels weten we dat onze elementen in een HTML-document injecteren via het <App>-component, ook beginnen met het maken van onze webpagina!

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 heet dit gerenderd.

###### *oefening*

app.js

import React from 'react';

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

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

import limoenen from './assets/limoenen.png';

import ijsblokjes from './assets/ijsblokjes.jpg';

import Product from './components/Product';

import './App.css';

function App() {

function logClick() {

console.log('You clicked!');

}

return (

<>

<nav>

<ul>

<li><a href="/">Shop</a></li>

<li><a href="/">Ons verhaal</a></li>

<li><a href="/">Blog</a></li>

</ul>

<ShoppingCart className="shopping-cart-icon"/>

</nav>

<header>

<h1>Fruit perfection</h1>

<button type="button" onClick={logClick}>

Shop nu

</button>

</header>

<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."

/>

<Product

image={limoenen}

title="Limoen"

description="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."

/>

<Product

image={ijsblokjes}

title="Limoen"

description="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."

/>

</main>

</>

);

}

export default App;

import React from 'react';

function Product(props) {

return (

<article className="product">

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

<h2 className="product-name">{props.title}</h2>

<p className="product-description">{props.description}</p>

</article>

)

}

export default Product;

Hoewel 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 gebruik** className en niet class.

**HTML:** <article **class**="product">

**React:** <article **className**="product">

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

#### Fragments(<> </>)

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 omwikkelende tag zetten, anders crasht onze applicatie.

**Waarom?**

 App is een functie. Je kunt volgens de regels van JavaScript niet twee verschillende dingen tegelijk teruggeven uit een functie, binnen één return statement.

We moeten de elementen omwikkelen (oftewel *wrappen*

gezien we geen extra styling gaan plaatsen, zorgt dit voor een onnodig, extra element. Hier heeft React iets op bedacht: **fragments**, deze worden gebruikt om het *één-return-value*-probleem op te lossen

function App() {

return (

<>

<div>Eerste element</div>

<div>Tweede element</div>

</>

);

}

#### Eigen afbeeldingen en iconen gebruiken

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

<img src="./assets/citroenen.jpeg" alt="Citroenen” />

Fout!

In React project moet **iedere bron eerst bovenaan in het bestand geïmporteerd worden**,anders kan het niet worden uitgelezen.

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

Let erop dat je geen hoofdletters (wel snake\_case) gebruikt in de import-naam.

#### 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, zodat je dat je oneindig ver in kunt zoomen.

Afbeelding met cirkel, Hemellichaam, Amber, astronomie

Automatisch gegenereerde beschrijving

Een SVG wordt vaak gebruikt voor bedrijfslogo's en icoontjes.

Laten we eerst de juiste elementen klaarzetten:

* Plaats een <nav>-tag *boven* de header met daarin:
* Een unordered list <ul>, met daarin:
* drie items <li>...
* … die elk een anchor tag <a>-bevatten. De link mag voor nu gewoon naar onze home-pagina wijzen ("/"). Zet hierin " Shop", "Ons verhaal" en "Blog".

Het winkelwagen-icoon willen we achter de unordered list in de navigatiebalk plaatsen. Maar hoe doen we dit? In de \* assets\*-map hebben we dit icoon al (winkelmandje.svg) voor je klaar staan.

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

Het is belangrijk om *hoofdletters* te gebruiken bij het ReactComponent en bij de gekozen naam.

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.

Onthoud dat wanneer je met SVG-bestanden werkt, je altijd de hoogte- en breedte properties toewijst in CSS!

#### Event listeners

We hebben inmiddels een prachtige button in onze header staan, maar deze button doet nu nog niets.

* 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 (

{/\* 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!');

}

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 diezelfde manier kunnen we onSubmit-events op formulieren plaatsen of onChange-events op invoervelden.

<button

type="button"

onClick={() => console.log("Jij wil shoppen!")}

>

Shop nu

</button>

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 geen aparte functie hebt - plaats de code dan in een anonieme functie:

#### Verdieping: imports en exports

*Code splitting* is in ieder soort project een belangrijk onderdeel, vooral in grotere projecten met React. Een gemiddeld stukje software bevat simpelweg te veel regels code om alles in één bestand te zetten:

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/import** en **named exports/export**,

###### **default exports**

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.

function sum(a, b) {

return a + b;

}

export default 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 Je mag de import een andere naam geven,

###### **named exports en import**

Soms exporteren we niet één, maar meerderde dingen uit één bestand.   
Je kunt slechts één onderdeel exporteren als standaard,

maar hebt daarna oneindig veel *named exports* voor alle andere onderdelen.

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.

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 wél de originele namen aanhouden.   
Dit is de enige manier om aan te duiden welke waardes 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? import { radius as banaan } from './helpers/calculateFunctions';

Hiermee zeggen we: importeer de *named export* radius, maar gebruik radius onder de naam banaan.

#### 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

##### Twee cruciale verschillen

* 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.

##### Herbruikbare react componenten

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 hergebruiken.

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:

Afbeelding met tekst, handschrift, Lettertype, schermopname

Automatisch gegenereerde beschrijving

##### Properties

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.

deze sliders is namelijk nagenoeg identiek, met een paar hele kleine verschillen:

* De volume slider heeft een volume-label en een range van 0 tot 11;
* De bass slider heeft een bass-label en een range van 0 tot 4;
* De treble slider heeft een treble label en een range van 1 tot 5;

waardes 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

function App() {

return (

<Slider textLabel="volume" minRange="0" maxRange="11"/>

<Slider textLabel="bass" minRange="0" maxRange="4"/>

<Slider textLabel="treble" minRange="1" maxRange="5"/>

);

}

###### props

Alle properties die wij meegeven worden verzameld in een object. Het Slider-component ontvangt alle properties, dit doen we door de parameter props

function Slider(props) {

}

###### Keys

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;

<https://vimeo.com/743053497/ba0ed6db2a>

#### Speciale props doorgeven: children

Het doorgeven van properties is ontzettend handig, maar het kent ook limitaties.

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 telkens anders(afbeeldingen, of een combinatie daarvan).

Afbeelding met tekst, elektronica, schermopname, Webpagina

Automatisch gegenereerde beschrijving

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:

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>

);

}

https://vimeo.com/743330257/54ff537781

#### Interactieve componenten en formulieren

##### State

Tot nu toe hebben we alleen statische pagina's gemaakt met React:

je weet nu hoe je werkt met afbeeldingen, tekst en herbruikbare componenten. Oké, bij het aanklikken van een button wordt er misschien iets in de console gelogd we kunnen nog niet spreken van een echte *applicatie*.

https://edhub.novi.nl/study/courses/516/content/12735

bij het schrijven van software 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 continue op verschillende plekken nodig Wanneer je dit goed wil organiseren, loop je al snel tegen een hindernis aan:

we willen onze applicatie zo veel mogelijk opsplitsen in verschillende componenten en features die verschillende taken uitvoeren. De data stroomt ongecontroleerd door onze applicatie:

Afbeelding met tekst, diagram, schermopname, Lettertype

Automatisch gegenereerde beschrijving

niet goed managen, is de data *overal(Single Source of Truth),* 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

Afbeelding met tekst, schermopname, Lettertype, Materiaaleigenschap

Automatisch gegenereerde beschrijving

Zie het als een gedeelde kantoorruimte: je werkt daar het liefst op twee beeldschermen, maar niet alle beeldschermen zijn voorzien van een HDMI-kabel,

Zonder State zul 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.

State gebruiken, kun je iedere ochtend gewoon naar de balie lopen om een kabeltje te vragen. De balie heeft de kabels in beheer en weet precies waar ze zijn,.

In de State slaan we data op die *veranderd*, bijvoorbeeld de hoeveelheid items in een winkelmandje.

Om waardes naar de State te schrijven en uit te lezen, gebruiken we speciale methodes genaamd **hooks**.

###### hooks

Hoe werkt dit precies?

Stel dat we een spel spelen met meerdere spelers, waarbij de gebruiker iemand 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 daarom opslaan in de State.

Afbeelding met tekst, Lettertype, schermopname, nummer

Automatisch gegenereerde beschrijving

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 **initiële waarde**

Indien je met getallen werkt, begin je vaak met 0, met arrays gaat werken, begin je met een lege array ([ ])

Afbeelding met tekst, Lettertype, schermopname, roze

Automatisch gegenereerde beschrijving

krijgen we een array terug met twee waardes 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 wil.
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.

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,

* set<naam van de State key> of
* toggle<naam van de State key>

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>

</>

);

}

<https://vimeo.com/744158534/3ae02d7c17>

###### Condities

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.

We willen het h1-element dus **conditioneel renderen**. Hiervoor zullen we de logica operatoren gebruiken die je eerder tijdens de cursus JavaScript hebt gebruikt: [en](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND) (&&) en [niet](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_NOT) (!) (zie ' Beslissingsstructuren' in JavaScript Hfst. 2).

// 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 de gebruiker vervolgens een button aanklinkt 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?

Hier kunnen we een **ternary operator** (conditie ? true : false) voor gebruiken:

// Impliciet gedefinieerd: bevat player een truthy waarde?

{player ? <h1>{player} is aan de beurt</h1> : <p>Er is nog niemand gekozen.</p>}

// Expliciet 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 ? plaats je de conditie die getoetst wordt
* Als de conditie *waar* is, wordt de eerste waarde getoond
* Als de conditie niet waar is, wordt de tweede waarde (na de :) getoond

Afbeelding met tekst, schermopname, roze, Lettertype

Automatisch gegenereerde beschrijving

###### Conditioneel stylen

Net als dat we elementen conditioneel kunnen renderen, kunnen we ook CSS-classes toewijzen op basis van condities.

Afbeelding met tekst, Lettertype, schermopname, logo

Automatisch gegenereerde beschrijving

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>

###### Conditionele properties

Op basis van diezelfde state, zullen we wellicht ook andere properties of attributen willen doorgeven.

willen we de 'Bart'-button bijvoorbeeld *disablen* als Bart op dat moment al aan de beurt is. Het button-attribuut disabled verwacht de boolean waarde true of false.

<button

type="button"

onClick={() => setPlayer('Bart')}

className={player === 'Bart' ? 'active' : 'default'}

disabled={player === 'Bart'}

>

Bart

</button>

<https://vimeo.com/744482094/1df03ae377>

#### Props vs. Callback props

##### callbacks

In het vorige hoofdstuk heb je geleerd hoe je componenten properties kunt meegeven.

Dit waren tot nu toe altijd primitieve 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(klikken op een button-component, of de functie die nodig is om validatie op een input-component toe te passen,).

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

});

een anonieme (ofwel: naamloze) arrow function () => {}

een gewone functie

array.map(doThings);

function doThings() {

// dit is de callback, waarin we telkens nieuw gedrag kunnen implementeren

}

Je hebt dus al heel veel met callbacks gewerkt

functies willen meegeven aan onze componenten, doen we dit op dezelfde manier als het doorgeven van normale properties.

Afbeelding met tekst, Lettertype, schermopname, logo

Automatisch gegenereerde beschrijving

js

// Button.js

function Button({ playerName, handlePlayerChange }) {

return (

<button

type="button"

onClick={() => handlePlayerChange(playerName)}

>

{playerName}

</button>

);

}

react

export default Button;

// 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.

#### Controlled components

Toen je net begon met HTML

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

Met één regel HTML had je een volledig functionerend wachtwoord-veld: de gebruiker kan input intoetsen, input plakken of deze weer weghalen en bovendien wordt de tekst niet weergeven in karakters, maar in bolletjes! Al deze gratis functionaliteit krijg je van de DOM,

Wilde je de ingevulde waardes uitlezen met JavaScript? Dan deden we dat zo:

const emailField = document.getElementById('email-field');

console.log(emailField.value);

###### uncontrolled components

We noemen standaard invoervelden daarom uncontrolled components: wij doen niets en de DOM regelt alles.

###### controlled components

invoervelden in React bouwen, zetten we invoervelden (meestal) op als controlled components: we overschrijven het standaard gedrag van de DOM met onze eigen logica,

###### Maar hoe werkt dit?

function App() {

const [emailValue, setEmailValue] = useState('');

return (

<input type="email"/>

);

}

Vervolgens moeten we zorgen dat de inhoud van ons tekstveld gelijk staat aan de inhoud van de state.

<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.  
het wordt een soort *loop*.

Afbeelding met tekst, diagram, schermopname, cirkel

Automatisch gegenereerde beschrijving

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)}

/>

);

}

e.target.value). Die waarde gebruiken we als argument voor onze *setter functie*, zodat we de emailValue state variabele kunnen updaten.

<https://vimeo.com/480909136/25118e85b2>

#### Meerdere inputs met één onChange-handler

##### onChange

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 herhalen:

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 [agreeNewsletter, toggleAgreeNewsletter] = useState(false);

<input

type="text"

name="firstname"

value={firstname}

onChange={(event) => setFirstname(event.target.value)}

/>

Gelukkig is er ook een manier om één onChange handler te schrijven die dit afhandelt

##### Een enkel input veld

in plaats van een losse state variabele, één state-**object** voor initialiseren.

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.

in de toekomst namelijk hergebruiken voor alle inputs in dit formulier. Omdat we nu nog maar één invoerveld hebben,

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 intoetst, wordt dit opgeslagen in de state.

##### 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.  
functie aanroepen bij een toetsaanslag in zowel het achternaam- als voornaam-veld,

Een fijne bijkomstigheid is dat het event-object dat bij iedere toetsaanslag gegenereerd wordt,

naast de ingevulde waarde óók de naam van het invuldveld registreert. Dit kunnen we terugvinden op e.target.name.

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?

een object-key geen punten mag bevatten.

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.

###### Spread

Hoe zorgen we er voor dat de huidige waardes van het object bewaard blijven,

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 onderaan 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.

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-select" opties zijn (slechts één antwoord mogelijk),

moeten alle name-attributen van radio-inputs die onderdeel zijn van zelfde vraag, identiek zijn.

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.

dus in onze handleChange-functie moeten controleren welk *type* invoer-veld op dat moment getriggerd wordt - en als dat een checkbox is, kiezen we voor de checked waarde:

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 ingetoetste waarde,

### Bonus: React Hook Form

Je hebt 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 invoervelden, 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 Hfst. 3.8.

wanneer je form-validatie op al deze velden wil gaat toepassen, of complexere formulieren wil gaan bouwen, blijft deze oplossing vrij arbeidsintensief.

een controlled component vraagt ook heel veel processorkracht:

##### librarys

heel belangrijk is dat je weet hoe je met de basis-functionaliteit van React een formulier opbouwt, vóór je aan de slag gaat met libraries en plugins.

Wanneer je grote, complexe formulieren nodig hebt in je applicatie is het daarom raadzaam om een library te gebruiken( [Formik](https://formik.org/" \t "_blank) , [Redux Form](https://redux-form.com/8.3.0/" \t "_blank) of [React Hook Form](https://react-hook-form.com/" \t "_blank)).

React Hook Form is de meest compacte en lichtgewicht library van de drie, die we gemakkelijk in ons project kunnen installeren via npm.

* gemakkelijker is om validatieregels toe te passen en we minder code hoeven te schrijven.
* minimaliseert React Hook Form het aantal re-renders

###### Installatie

Om React Hook Form te implementeren, zullen we het eerst als dependency moeten installeren

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 exporteert, kijk je altijd eerst even in de [documentatie](https://www.npmjs.com/package/react-hook-form)!

<https://www.npmjs.com/package/react-hook-form>

De useForm-methode geeft toegang tot alle functionaliteiten die deze veelzijdige package ons biedt.

##### Invoervelden registeren

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>

);

}

###### Registratie

Useform

jsx

import React from 'react';

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

function MyForm() {

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

const onSubmit = (data) => {

console.log(data);

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<label>

Name:

<input type="text" {...register('name', { required: true })} />

{errors.name && <span>This field is required</span>}

</label>

<label>

Email:

<input type="email" {...register('email', { required: true, pattern: /^\S+@\S+$/i })} />

{errors.email && <span>Please enter a valid email address</span>}

</label>

<label>

Message:

<textarea {...register('message')} />

</label>

<button type="submit">Submit</button>

</form>

);

}

export default MyForm;

In dit voorbeeld wordt de useForm-hook geïmporteerd uit de react-hook-form-bibliotheek. De hook wordt gebruikt om formulierfunctionaliteit toe te voegen aan het component. De hook retourneert een aantal waarden, waaronder register, handleSubmit en formState.

In React kan de useForm-hook worden gebruikt om formulieren te beheren en de staat van formulierelementen bij te houden. De useForm-hook is een handige manier om formuliervelden te initialiseren, hun waarden bij te werken en formuliergegevens te verzamelen wanneer het formulier wordt verzonden.

Hier is een voorbeeld van hoe je de useForm-hook kunt gebruiken in een React-functioneel component:

jsx

import React, { useState } from 'react';

function MyForm() {

const [formState, setFormState] = useState({

name: '',

email: '',

message: '',

});

const handleChange = (e) => {

setFormState({

...formState,

[e.target.name]: e.target.value,

});

};

const handleSubmit = (e) => {

e.preventDefault();

// Verwerk het formulier of verzamel de gegevens

console.log(formState);

};

return (

<form onSubmit={handleSubmit}>

<label>

Name:

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

</label>

<label>

Email:

<input type="email" name="email" value={formState.email} onChange={handleChange} />

</label>

<label>

Message:

<textarea name="message" value={formState.message} onChange={handleChange} />

</label>

<button type="submit">Submit</button>

</form>

);

}

export default MyForm;

In dit voorbeeld wordt de useForm-hook niet specifiek gebruikt, maar in plaats daarvan wordt de useState-hook gebruikt om de staat van het formulier bij te houden in de formState-variabele. De formState-variabele bevat een object met de velden name, email en message, en hun bijbehorende waarden worden bijgewerkt met behulp van de setFormState-functie.

De handleChange-functie wordt opgeroepen wanneer er een wijziging plaatsvindt in een formulierveld. Deze functie gebruikt de spread operator om de huidige formState te kopiëren en de waarde van het gewijzigde veld bij te werken op basis van de e.target.name en e.target.value.

De handleSubmit-functie wordt opgeroepen wanneer het formulier wordt verzonden. Hier kun je de logica toevoegen om het formulier te verwerken of de verzamelde gegevens te gebruiken.

In de JSX-rendering worden de formuliervelden gemaakt met de juiste name-attributen en worden hun waarden gekoppeld aan de formState-waarden. De onChange-attribuut wordt gebruikt om de handleChange-functie aan te roepen wanneer er een wijziging plaatsvindt in een formulierveld. De onSubmit-attribuut van het <form>-element verwijst naar de handleSubmit-functie om het formulierinzendingen te verwerken.

Dit is een eenvoudig voorbeeld om de basisprincipes te demonstreren van het gebruik van een formulier met behulp van React. Je kunt deze code aanpassen en uitbreiden op basis van je specifieke formuliervereisten.

register wordt gebruikt om elk formulierveld te registreren, waarbij de naam van het veld en eventuele validatieregels worden opgegeven.

handleSubmit wordt gebruikt om de inzending van het formulier te verwerken, waarbij de bijbehorende submit-functie wordt aangeroepen.

formState bevat informatie over de status van het formulier, zoals eventuele fouten tijdens de validatie.

Binnen de form-tag worden de formuliervelden gemaakt met behulp van reguliere HTML-input-elementen. Het register-attribuut wordt toegepast op elk formulierveld om het te registreren met de useForm-hook, inclusief eventuele validatieregels.

De onSubmit-functie wordt aangeroepen wanneer het formulier wordt verzonden. Hier kun je de logica toevoegen om het formulier te verwerken of de verzamelde gegevens te gebruiken. In dit voorbeeld wordt de ingediende data simpelweg gelogd.

De errors-object wordt gebruikt om eventuele foutmeldingen weer te geven op basis van de validatieregels die zijn opgegeven in het register-attribuut. Als er een fout is voor een bepaald veld, wordt de foutmelding weergegeven.

Eén van de methodes die je altijd nodig hebt, is register.

invoervelden die we willen gaan beheren met React Hook Form namelijk altijd eerst **registreren**.

raadzaam deze methode direct te destructuren uit de useForm-aanroep:

function App() {

const { register } = useForm();

return (

{/\* formulier... \*/ }

);

}

We gebruiken deze methode om ieder 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

De register-methode plaatst dus zowel een referentie, als een naam

Dat zijn dus meerdere attributen in één, waardoor we de **spread operator** (...) moeten gebruiken:

// VOOR

<input

type="text"

name="email"

/>

// NA

<input

type="text"

{...register("email")}

/>

Bovenstaand input-element(// NA) is nu onderdeel van het React Hook Form, onder de naam email.

Daarmee wordt het hele verhaal met onChange-listeners en value-attributen in één keer voor ons geregeld.

#### Implimentatie van React Hook Form

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="4"

cols="40"

placeholder="Laat je bericht achter"

{...register("message-content")}

>

</textarea>

</label>

<button type="submit">

Versturen

</button>

</form>

);

}

#### Submitten

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 waardes verzamelen!');

}

return (

<form onSubmit={handleFormSubmit}>

{/\* input velden... \*/}

</form>

)

}

Op de <form>-tag staat een onSubmit event-listener die wordt getriggerd zodra er er op een submit-button binnen de <form>-tag wordt geklikt,

functie die dan wordt uitgevoerd, hebben we handleFormSubmit genoemd. Die zorgt er met preventDefault() voor dat de pagina niet ververst 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 als het ware verreikt met React Hook Form functionaliteit.

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.

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",

message-content: "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 eentje? Dan ben je dat invoerveld waarschijnlijk vergeten te registreren.

#### Validatie regels

Eén van de voordelen van React Hook Form is dat het valideren van de velden ontzettend makkelijk te implementeren is.

Stel dat we het invoerveld voor het bericht van de gebruiker verplicht willen maken, maar ook willen afdwingen 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("message-content", {

required: true,

minlength: 10,

maxLength: 50,

})}

></textarea>

Deze drie regels worden gecontroleerd zodra de gebruiker op "verzenden" drukt.

een standaard set validatieregels beschikbaar die je kunt meegeven:

* *Required* - voor verplichte velden (uitgedrukt 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-getal-letter-letter*) of gebruik van speciale tekens (als RegEx patroon);

<https://react-hook-form.com/docs/useform/register>

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.

Als je wel benieuwd bent naar RegEx en hier meer over wil weten, zijn hier genoeg tutorials over te vinden, zoals bijvoorbeeld [deze.](https://www.youtube.com/watch?v=rhzKDrUiJVk)

#### Foutmeldingen toevoegen

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

Om foulmeldingen toe te kunnen voegen, geven we iedere regel een eigen object:

<textarea

{...register("message-content", {

required: {

value: true,

message: 'Dit veld is verplicht',

},

minLenght: {

value: 10,

message: 'Input moet minstends 10 karakters bevatten',

},

maxLenght: {

value: 50,

message: 'Input mag maximaal 50 karakters bevatten',

},

})}

></textarea>

Wanneer je een foutmelding wilt instellen voor de validate-regel, werkt dit net ietsje anders. Dan zul je een [of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_OR)-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,

eerste dat we daarvoor nodig hebben, is het errors object(uit de useForm-call) van React Hook Form.

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

Afbeelding met tekst, schermopname, software, Lettertype

Automatisch gegenereerde beschrijving

een key voor ieder veld dat op dat moment niet voldoet aan de validatieregels. De velden die goedgekeurd zijn, staan namelijk niet in dit error-object.

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

<https://vimeo.com/604635472/e068d6db75>

#### 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-methode](https://react-hook-form.com/api/useform/watch" \t "_blank) nodig

<https://react-hook-form.com/docs/useform/watch>

Laten we beginnen met een selectbox

In deze situatie gaan we er vanuit dat we deze elementen toevoegen aan een bestaand React Hook Form,

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")}

/>

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

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

Nu kun je een variabele aanmaken waarin het resultaat van de watch-methode wordt opgeslagen.

// Optie 1: houd de waarde van één veld bij

const watchSelectedReferrer = watch('found-through');

// Optie 2: houd de waarde van meedere velden bij:

const watchFields = watch(['found-through', 'naam-ander-veld']);

// Optie 3: houd de waardes van alle velden bij:

const watchAllFields = watch();

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>

);

}

#### Formulier opties

##### mode

Onze formulieren komt met een aantal standaardinstellingen. Gelukkig kunnen we die gemakkelijk wijzigen

Misschien wil je sommige invoervelden al vullen met waardes 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"

Het wachten met valideren tot de gebruiker het formulier probeert te verzenden, is standaard functionaliteit van React Hook Form.

andere mogelijkheden:

* onSubmit: pas valideren wanneer er op de submit-knop geklikt wordt (standaard);
* onBlur: valideren zodra een gebruiker een veld weer heeft 'verlaten';
* onChange: valideren terwijl de gebruiker aan het typen is, dus op ieder onChange-event. *Let op*: het formulier zal bij elke toetsaanslag opnieuw valideren, dus dit heeft impact op de perfomance 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;
* onTouched: valideren zodra een gebruiker een veld voor de eerste keer heeft 'verlaten', en daarna bij iedere 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 waardes, door de defaultValues instelling te overschrijven:

function App() {

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

mode: 'onBlur',

defaultValues: {

'found-through': 'advertentie',

age: 12,

},

});

}

Let erop 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!

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 [uitgebreide documentatie](https://react-hook-form.com/api/useform) van React Hook Form: deze is super overzichtelijk en altijd voorzien van duidelijke code-voorbeelden!

[useForm (react-hook-form.com)](https://react-hook-form.com/docs/useform)

#### Van elementen naar componenten

Wanneer je jouw invoervelden wil 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:

Afbeelding met tekst, schermopname, Lettertype, diagram

Automatisch gegenereerde beschrijving

In onderstaand voorbeeld werken we nog in App.js, maar in de toekomst zul je deze code waarschijnlijk op een aparte pagina ( ContactPage.js of Register.js).

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 handleFormSubmit 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="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>}

</>

);

}

export default InputComponent;

We gaan er daarom voor zorgen dat wanneer we ons component gebruiken in App.js

// App.js

<InputComponent

inputType="text"

inputName="name"

inputId="name-field"

inputLabel="Naam:"

validationRules={{

required: {

value: true,

message: 'Naam is verplicht',

}

}}

// 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 banaan={register} kunnen doen, maar op deze manier blijft de schrijfwijze hetzelfde wanneer we de register-methode gaan toepassen in het component.

Daarnaast zie je ook dat we bij het doorgeven van de properties de methode *niet aanroepen* en er ook geen argumenten, pas in het component zelf:.

Omdat we de register-methode onder de naam register hebben meegegeven, kunnen we hem net zo opschrijven als wanneer je geen componenten zou gebruiken.

Om de errors goed weer te geven, zullen we gebruik moeten maken van de **blokhaak-notatie** in plaats van de **punt-notatie**. We willen immers een variabele als object-key gebruiken:

// 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>

);

}

#### Routing

In dit hoofdstuk ga je alles leren over het toepassen 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 hoefden 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!  
De server ging dan op zoek naar de HTML-, CSS- en JavaScript voor die specifieke pagina, goot alles in één kant-en-klaar HTML-bestand en stuurde dat terug naar de browser.

Iedere keer als de gebruiker op een link klikte of een actie uitvoerde (zoals het versturen van een contact-formulier) werd er een nieuwe pagina opgevraagd en de hele applicatie ververst.

Met de komst van Frontend Frameworks werd Client Side Rendering geïntroduceerd:

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.

Vervolgens wordt de applicatie eenmalig opgebouwd in de browser: alle pagina's in één keer. Om deze reden noemen we React applicaties Single Page Applications(spa)

De React Router voor dat we alleen de pagina weergeven die correspondeert met de url in de adresbalk. Conditioneel pagina’s renderen.

Maar wanneer je op deze manier tussen pagina's wisselt, kom je voor een aantal uitdagingen te staan. Die uitdaging ligt ‘m niet per se in het laden van de content, maar in het bieden van een gebruikerservaring die overeenkomt wat gebruikers gewend zijn.

wanneer zij onze applicatie gebruiken, verwachten ze dat:

* De URL in de adresbalk overeenkomt met hetgeen wat ze op dat moment bekijken;
* Ze de browser-knoppen Vorige en Volgende kunnen gebruiken (... en dat er ook iets logisch gebeurt wanneer ze erop klikken);
* Ze rechtstreeks naar een specifieke weergave (ook wel bekend als [deep linking](https://en.wikipedia.org/wiki/Deep_linking" \t "_blank)) kunnen navigeren met de juiste URL;

Maar omdat een Single Page Application technisch gezien nooit daadwerkelijk van pagina of **url** veranderd, moeten we een aantal stappen ondernemen om dit gedrag na te bootsen.

Nu we onze applicaties gaan uitbreiden met meerdere pagina's, is het belangrijk om eerst naar de indeling van ons project te kijken.

*eigen* stylesheet te koppelen en deze samen in een mapje te stoppen:

src

└── components

└── button

├── Button.js

└── Button.css

└── header

├── Header.js

└── Header.css

Wanneer we pagina's gaan maken, zijn dit in feite ook gewoon componenten die ieder hun eigen stylesheet gebruiken.

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 ... \*/}

</>

);

}

[**https://vimeo.com/775072298/22e2eedb6c**](https://vimeo.com/775072298/22e2eedb6c)

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 package](https://www.npmjs.com/package/react-router-dom" \t "_blank) te gebruiken.

[**https://www.npmjs.com/package/react-router-dom**](https://www.npmjs.com/package/react-router-dom)

Wanneer je dit wil gaan toepassen in jouw project, zul je het altijd eerst even moeten installeren

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:

##### Routes definiëren

###### routes

Vervolgens kun je gaan definiëren welke URL's de gebruikers mogen bezoeken en, belangrijker nog, welke pagina's zij dan te zien krijgen.

Hiervoor gebruik je één <Routes> component, waarin je alle afzonderlijke <Route>-componenten definieert. Ieder <Route>-component verwacht twee properties:

1. Het **path**, die wijst naar de url. Hierin hoef je *nooit* de basis-url (zoals *http://localhost:3000* of *www.jouw-website.nl*) mee te nemen: dit doet React automatisch.   
   Je gebruikt dus alleen de relatieve url die daaraan vastgeplakt wordt, zoals / voor home, /contact voor de contact-pagina en \* voor alles dat niet overeenkomt met de bovengenoemde routes.  
   Uiteraard mag je iedere url kiezen die je wil! De FAQ-pagina op *http://localhost:3000/banaan*? Geen probleem. Alleen brengt dit jouw gebruikers waarschijnlijk wel een beetje in de war.   
   Jouw url’s duidelijk beschrijven wat er op de pagina staat en dat ze *nooit* hoofdletters of spaties bevatten.
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 menubalk... \*/}

<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 onze toekomstige menubalk buiten het <Routes>-component blijven staan: deze staat immers boven iedere pagina en hoeft niet te wissele  
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 géén navigatie of footer hebben en sommige wel? Of ziet de footer er op sommige pagina's anders uit? In dat geval zul je het navigatie- en footer element op iedere pagina afzonderlijk moeten invoegen,

##### Links

we hebben onze routes gedefiniëerd,

We willen de gebruiker daarom met linkjes 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.

Afbeelding met tekst, Lettertype, schermopname, wit

Automatisch gegenereerde beschrijving

// 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 nooit meer een anchor-tag mag gebruiken. De a-elementen zullen we alleen nog gebruiken als we naar webpagina’s buiten onze applicatie linken,

###### NavLinks

###### link

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.

Afbeelding met tekst, Menselijk gezicht, schermopname, Website

Automatisch gegenereerde beschrijving

###### Nalink

Om deze reden heeft React Router ook <NavLink>-componenten: deze zijn zich bewust van de huidige url en weten daarom wanneer ze actief zijn.

Om straks te kunnen wisselen,

Om straks te kunnen wisselen, kunnen we alvast twee classes in de CSS klaarzetten:

// het uiterlijk van een actieve link:

.active-menu-link {

border-bottom: none;

}

// het uiterlijk van een inactieve link

.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>

);

}

Op dit 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).

deze syntax ook gewoon "aannemen" zoals 'ie is:

<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 NaLink?**

Voor we ingaan op hoe React Router dicteert dat de conditionele styling moet worden toegepast, gaan we eerst eens kijken op hoe we dat op een normaal element zouden doen.

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 oplossen met een ternary operator:

// Navigation.js

function Navigation() {

const isActive = true;

return (

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

);

}

wil React Router dat je een anonieme functie meegeeft aan de className-property (een callback).

Deze anonieme functie krijgt dan *automatisch* een object toegeworpen waar deze informatie op staat - een beetje zoals het event-object wanneer je eventlisteners gebruikt.

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),   
óf 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'}>

NavLink ziet er zo uit:

<NavLink

className={({ isActive }) => isActive ? 'active-menu-link' : 'default-menu-link'}

to="/">

Home

</NavLink>

###### 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>

Hoe moeten we het oplossen wanneer we de gebruiker door willen sturen *nadat* een formulier ingevuld en verstuurd is?

Afbeelding met tekst, schermopname, Lettertype, ontwerp

Automatisch gegenereerde beschrijving

Net als bij state, 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.

#### Dynamische routes

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 eigen 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.

Hoeveel pagina's denk je dat we in React nodig hebben om al die producten weer te geven?

Precies: maar één.

Net als dat we componenten hergebruiken, zullen we dat ook doen met gelijknamige pagina's. Iedere keer als een gebruiker de pagina over sperziebonen (artikelnummer 395946) bezoekt, wordt de template-pagina gevuld met data over sperziebonen.

##### hoe weet de pagina nou welk product de gebruiker op dat moment bekijkt...?

###### dynamische parameters

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.

* www.ah.nl/product/395946 (sperziebonen)
* www.ah.nl/product/385831 (griesmeelpudding)

het laatste gedeelte van de url zal 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:

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.

###### userParams-hook

We moeten deze dynamische waarde dus weer kunnen opvragen uit de url doormiddel 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>

)

}

#### Beveiligde routes

Wanneer we een url beveiligen, betekent dit dat we dat we altijd eerst controleren of er wel voldaan wordt aan de conditie vóórdat we de pagina weergeven(gebruiker ingelogd moet zijn, of dat de gebruiker admin-rechten).

Wanneer dit niet zo is, willen de gebruiker dan netjes terugsturen naar de home- of inlogpagina.

Afbeelding met tekst, Lettertype, roze, ontwerp

Automatisch gegenereerde beschrijving

Deze conditie moet daarom *in* de route worden geëvalueerd:

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.

Deze conditie moet daarom *in* de route worden geëvalueerd:

return (

<Routes>

<Route path="/faq" element={/\* DE CONDITIE KOMT HIER \*/} />

</Routes>

)

<https://edhub.novi.nl/study/courses/516/content/15156>

Nu is het zo dat de element-property van het <Route>-component alleen elementen of componenten accepteert, zoals <Faq> of <p></p>.

##### Navigate

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.

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.

**Doordenkvraag:** 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/>}/>

Het stukje code dat je hebt gegeven, lijkt correct te zijn in termen van syntaxis, maar het kan problematisch zijn in termen van logica en leesbaarheid. Hoewel het technisch gezien werkt, kan het voor sommige situaties verwarrend zijn.

Het gebruik van de ternaire operator (**isLoggedIn === true ? <Faq/> : <Home/>**) in de **element**-prop van de **Route**-component bepaalt welk component wordt gerenderd op basis van de waarde van **isLoggedIn**. Als **isLoggedIn** waar is, wordt de **Faq**-component weergegeven, anders de **Home**-component.

Echter, het probleem met deze aanpak is dat het de leesbaarheid van de code kan verminderen, vooral als er meerdere voorwaarden zijn om te controleren. Het wordt al snel complex en moeilijk te begrijpen wanneer er meer dan één conditie is.

Een betere benadering zou zijn om de **Route**-component te splitsen en expliciet te definiëren voor elke mogelijke conditie. Bijvoorbeeld:

jsx

<Route path="/faq" element={isLoggedIn ? <Faq/> : <Home/>}/> <Route path="/faq" element={<Home/>}/>

#### API's en data requests (opdracht)

<https://github.com/hogeschoolnovi/frontend-react-weatherapp-tutorial/blob/stap-1-weersvoorspelling-ophalen/src/App.js>

<https://github.githistory.xyz/hogeschoolnovi/frontend-react-weatherapp-tutorial/blob/stap-1-weersvoorspelling-ophalen/src/App.js>

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.

verschillende typen requests

In de boilerplate zie je dat we al een aantal componenten voor je hebben opgezet, zoals de zoekbalk (<SearchBar />), de navigatie (<TabBarMenu />), een weer-blok (<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 modulair.

<https://github.com/hogeschoolnovi/frontend-react-weatherapp-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. Vergeet niet eerst een npm install te draaien zodat alle benodigde node modules worden geïnstalleerd. Daarna kun je de applicatie opstarten 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-blok (<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 modulair.

##### Axios request types

###### Try/catch

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 inmiddels 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 {

const result = await <insert-request-here>;

} catch(e) {

console.error(e);

}

}

###### Endpoint

###### Request body

###### **HTTP Header** /**request header**

Een request bevat altijd een **endpoint**, dit is het *adres* waarmee we de data van- of naar de server sturen, het geval dat we extra gegevens willen meesturen, stoppen we deze data in de **request body**.

daarnaast nog extra specificaties aan ons request willen toevoegen, doen we dit in de **HTTP Header** van het request (ookwel request header genoemd)

###### 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,

1. Het endpoint waarvandaan we de data willen ophalen (*verplicht*)
2. Een configuratie object met eventuele extra specificaties (*optioneel*)

axios.get(url [, config])

Dit configuratie-object is de request header. Dit object bevat extra data over het request

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,

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 authorisatie-token meestuurt.

###### POST-requests

Een POST-request is altijd bedoeld om *data te versturen*, zoals bij het registeren 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

1. Het endpoint waarnaartoe we de data willen versturen (*verplicht*)
2. Het data-object met alle gegevens die we willen verzenden(request body) (*verplicht*)
3. Een configuratie object met eventuele extra specificaties (*optioneel*)
4. axios.post(url, data [, config])

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,

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 authorisatie-token meestuurt.

###### PUT- en DELETE-requests

* Een PUT-request is bedoeld om bestaande data te wijzigen.
* Een DELETE-request is bedoeld om bestaande data te *verwijderen*.

heeft een PUT-request dezelfde syntax als een POST-request.

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

een DELETE-request dezelfde syntax als een GET-request.

axios.delete(url [, config])

###### 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.

In de cursus JavaScript heb je geleerd dat API's interactie mogelijk maken tussen verschillende applicaties door de overdracht van gegevens van systeem naar systeem te faciliteren. Een API is de "ober" die zorgt dat jouw bestelling bij de keuken komt en jouw gerecht weer bij jou.

OpenWeather heeft vastgesteld dat we op de volgende manier gegevens in de database mogen opvragen 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=nl

Zoals je ziet, zit er een hoop informatie in dit endpoint verscholen.

stuurt ons een groot JSON object terug

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 te voorkomen.

De API-key fungeert vaak als een unieke identificatiemethode én als een geheime token voor authenticatie.

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.

heb je een API-key nodig. Deze kun je gemakkelijk verkrijgen door jezelf aan te melden bij OpenWeather(we zullen gebruik maken van het gratis "plan").

https://home.openweathermap.org/users/sign\_up

Als je hierna naar de tab “API keys” gaat, kun je hier jouw persoonlijke API key vinden. Die heb je straks nodig, dus hou hem even bij de hand.

Afbeelding met tekst, schermopname, Lettertype, software

Automatisch gegenereerde beschrijving

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!

Maak bovenaan in het bestand App.js een variabele apiKey aan en stop jouw API key daarin.

const apiKey = 'd85e6262984729404acc4eb353f4';

function App() {

// …

}

(opdracht)

#### Data fetchen

We weten nu dat we door middel van een specifieke url gegevens kunnen opvragen 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 variabele axios gebruiken als basis om de verschillende request-methodes die deze library beschikbaar heeft gesteld, te gebruiken. Gezien wij weersvoorspellingen 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 OpenWeather API om het weer voor Utrecht op te vragen. Gebruik de GET link uit de vorige paragraaf en vervang de {NAAM STAD} door utrecht en {JOUW API KEY} door jouw apiKey variabele. Tip: met string interpolation is dit makkelijker!
* Deze functie wordt nog niet afgevuurd (dus er zal ook nog niets gebeuren). Zorg wel alvast dat je de response value in de console logt zodat we straks kunnen checken of het request gelukt is.

import axios from 'axios';

const apiKey = 'd85e6262984729404acc4eb353f4';

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);

}

}

}

###### String interpolation

In JavaScript kun je string interpolatie bereiken met behulp van template literals, die worden gedefinieerd tussen backticks ( ``). Variabelen of expressies kunnen worden opgenomen in de string door ze tussen ${} te plaatsen.

Zie je hoe we string interpolation hebben gebruikt in de API url en zo de variabele apiKey erin gezet

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 aangeklikt;

function App() {

return (

<button

type="button"

onClick={fetchData}

>

Haal data op!

</button>

);

}

Wanneer je de console opent in Google Chrome (Windows: Ctrl + Shift + J, Mac: option + CMD + J) en dan op de knop drukt, zie je daar een data object verschijnen:

Afbeelding met tekst, software, nummer, Lettertype

Automatisch gegenereerde beschrijving

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';);
* Creeër vervolgens de state variabele weatherData en een bijbehorende setter-functie;
* Roep na de console.log() de setterfunctie aan en geef daar de response aan 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....`);

console.log(result.data);

setWeatherData(result.data);

} catch (e) {

console.error(e);

}

};

}

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)

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 className="location-details">

<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>

);

}

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

TypeError: Cannot read property 'weather' of undefined

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 primitief datatype was (string, boolean, nummer, null of undefined)

{/\*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 evalueren naar truthy en krijgen we alsnog een foutmelding.

Wanneer we structurele datatypes gebruiken (zoals objecten en arrays) zullen we moeten checken of er wel keys of waardes in staan, voor we elementen renderen:

function App() {

return (

<span className="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.

Omdat de temperatuur in Kelvin wordt aangeleverd lijkt het nu alsof het 304.05 graden is in Utrecht... Warm! Dit gaan we later converteren.

#### 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 input-element om moeten zetten naar een controlled component.

* Importeer useState bovenaan het bestand.
* Maak een stukje state aan voor de variabele query (een zoekterm noemt men een search query in het Engels) en wijs je een lege string als initiële 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 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 zoek-knop óf op Enter drukt, willen we dus iets gaan doen met de query.

* Vervang het span-element voor een form-element;
* 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 refresht 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 locatie bevat (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 dáár hebben we hem nou juist nodig, want daar wordt de data opgehaald...

We hebben dus een manier nodig om vanuit <SearchBar /> een stukje state 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 dus wél kunnen doen, is de functie die de state in <App /> aanpast, doorgeven aan <SearchBar />! Dat doen we via callback props!

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

function App() {

const [location, setLocation] = useState('');

return (

// ... de rest

)

}

Dan kunnen we die setLocation functie nu in z’n geheel als property 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 (in SearchBar.js) zodat we properties kunnen ontvangen - óf destructure setLocationHandler al direct.
* Je kunt de setLocation functie nu gebruiken middels props.setLocationHandler() of setLocationHandler als je deze gedestructured hebt.
* Roep deze functie aan in de onFormSubmit 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.

Ziet 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>

)﻿

}

Nu krijgt App.js telkens een nieuwe locatie beschikbaar in de state als de gebruiker in <SearchBar /> op ‘zoeken’ klikt of op enter drukt.

Deze locatie zullen we straks gebruiken om de 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 aangeroepen wanneer de gebruiker op de ‘haal data op!’ knop drukt, maar wanneer de variabele location veranderd is.

#### Component Life Cycles

Elk 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.

wanneer state-variabelen geüpdatet 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.*

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 (

//...

);

}

###### useEffect-hook

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

**Wanneer gaan we dit gebruiken?**

* Wanneer we externe data ophalen. Dit willen we *eenmalig* doen, namelijk zodra de gebruiker een pagina *opent*. 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 aangezwengeld of uitgaande netwerk requests die nog niet klaar waren.
* Wellicht hebben we grafische elementen in een canvas <element>-gegenereerd, en moet dit in real-time aangepast worden zodra we andere waardes binnen krijgen.
* Het komt vaak voor dat we data willen ophalen of versturen, maar *pas wanneer een specifieke waarde in de state geüpdatet is*.

Wanneer je begrijpt hoe de levenscyclus van een component eruit ziet,

op welk moment gebeurt. Deze cycli kunnen we categoriseren in 3 (oké, 3 en een beetje) stadia:

1. Initialiseren (*state* wordt aangemaakt, maar dit telt niet echt als life cycle)
2. Mounting (het component bijvoegen in de DOM, ofwel: “renderen” op de pagina)
3. Updating (wanneer de properties veranderen of het component aangepast wordt)
4. Unmounting (het component uit de DOM verwijderen en vernietigen)

###### een side effect

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.

##### De mounting cycle

De **mounting cycle** is het moment nadat ons component *voor het eerst is gerenderd*

De gebruiker vraagt de productpagina op door naar www.ah.nl/product/385831 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 die we aan al onze componenten willlen doorgeven(gebruiken we de useEffect-functie van React).

Deze functie krijgt altijd twee parameters mee:

1. Een **callback functie** met daarin de code die moet worden afgevuurd op dat specifieke moment: () => {}
2. Een **dependency array** die aangeeft *wanneer* (na welke life cycle) de code wordt uitgevoerd: [].

useEffect(() => {

//... voer dingen uit

}, []);

de dependency array, is erg belangrijk voor de werking van useEffect

drie opties:

* Een lege dependency array ([]): de callback wordt uitgevoerd na *mounting*
* Een dependency array met één of meerdere waardes: [state variabele, properties]: de callback wordt uitgevoerd nadat één van de meegegeven waardes is veranderd, na *update*.
* Géén dependency array: de callback wordt uitgevoerd *op iedere re-render* (tijdens iedere fase).

Het ophalen van data willen we één keer uitvoeren.

In dit geval geven we de useEffect-functie 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('Je ziet mij alleen op refresh');

}, []); // <---- de lege dependency-array staat hier

return (

<h1>Product</h1>

);

}

##### De update cycle

De **update cycle** zorgt ervoor dat we acties kunnen uitvoeren die afhankelijk zijn van het updaten van andere waardes.

De code die wij dan willen uitvoeren, zou je een soort *side effect* noemen.

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.

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>

</>

);

}

Deze specifieke useEffect-functie wordt iedere keer afgevuurd nadat de waarde in count2 is aangepast.

Side note: wanneer dit component voor het eerst wordt gerenderd, zul 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 initiële waarde (0) wordt veranderd, ook telt als een update.

##### De unmount cycle

Als een component niet meer weergegeven wordt, wil je ook dat de onderliggende processen stoppen, zoals timers, intervallen

De useEffect-functie is zo geschreven dat we een **clean-up functie**mogen returnen 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

Memory leak

We gebruiken deze cleanup-functie om lopende requests te stoppen, mocht de gebruiker in de tussentijd ineens 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, terwijl de asynchrone functie nog aan het wachten is op de response van de server.

De state-setter functie die de ontvangen data in de state zet, wordt vervolgens ook nog aangeroepen. Hierdoor ontstaat de volgende foutmelding:

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.  
De applicatie crasht dus niet, maar wordt op ten duur zeer traag.

**abortcontroller**

signal

Om onze requests in dit geval te kunnen annuleren, hebben we een axios **abortcontroller** nodig. Eerst maken we met axios 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 weggelaten om de illustratie simpeler te houden.

cleanup

abort

Nu 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 (

//....

);

}

Intervallen opruimen

Memory leak

setInterval

setTimeOut

clearInterval

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)

Deze setInterval-functie willen we nooit direct 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 intervallen die nog bezig waren.

We willen deze functie *eenmalig* starten, *nadat het component voor het eerst gemount is*.

function ProductPage() {

useEffect(() => {

setInterval(() => {

console.log("loading...");

}, 1000);

}, []);

return (

<p>Open de browserconsole om de intervallen te zien!</p>

);

}

We willen er echter ook voor zorgen dat wanneer dit component of deze pagina niet meer actief is, *dat ook het interval stopt*.

Als we dit niet expliciet benoemen, zal het interval namelijk gewoon doorlopen 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 voor 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>

);

}

useEffect in de praktijk

https://edhub.novi.nl/study/courses/516/content/12760