HO
GENT

Faculteit Bedrijf en Organisatie

Een vergelijking tussen Mithril.js, Angular en React

Jelle Callewaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Lotte Van Steenberghe
Co-promotor:
TBD

Instelling: —

Academiejaar: 2019-2020

Eerste examenperiode

Faculteit Bedrijf en Organisatie

Een vergelijking tussen Mithril.js, Angular en React

Jelle Callewaert

Scriptie voorgedragen tot het bekomen van de graad van
professionele bachelor in de toegepaste informatica

Promotor:
Lotte Van Steenberghe
Co-promotor:
TBD

Instelling: —

Academiejaar: 2019-2020

Eerste examenperiode

Woord vooraf

Samenvatting

Angular, React en Mithril zijn drie redelijk jonge[1] frameworks of bibliotheken die een developer kunnen helpen bij het maken van een SPA[2].

In deze thesis wordt onderzocht welke zaken belangrijk zijn bij het maken van de keuze tussen Angular, React of Mithril. Dit is dan ook de onderzoeksvraag. Om op deze vraag een duidelijk antwoord te kunnen geven, worden enkele kleinere deelvragen beantwoordt:

- Wat zijn de gemeenschappelijke aspecten tussen Angular, React en Mithril?
- Wat zijn de voornaamste verschillen tussen Angular, React en Mithril?
- Welk framework of bibliotheek biedt de hoogste performantie[3]?
- Welk is het meest populaire[4] framework of bibliotheek?

Als eerste werden Angular, React en Mithril grondig bestudeerd. Hierdoor kon een algemeen beeld gevormd worden over de drie technologieën[5].

Daaropvolgend werden enkele testgevallen opgesteld en uitgevoerd om te bepalen welk van de drie de beste performantie biedt[6].

Als laatste werden de technologieën met elkaar vergeleken om zo tot een oplossing te komen op de onderzoeksvraag[7].

[1] Angular werd uitgegeven in 2016, React in 2013 en Mithril in 2014
[2] Single Page Application
[3] Hierbij worden begrippen als snelheid en geheugenverbruik onderzocht.
[4] Gebaseerd op npm trends, Github statistieken en StackOverflow populariteit.
[5] Hoofdstuk 2
[6] Hoofdstuk 3
[7] Hoofdstuk 4

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

1. Inleiding

De inleiding moet de lezer net genoeg informatie verschaffen om het onderwerp te begrijpen en in te zien waarom de onderzoeksvraag de moeite waard is om te onderzoeken. In de inleiding ga je literatuurverwijzingen beperken, zodat de tekst vlot leesbaar blijft. Je kan de inleiding verder onderverdelen in secties als dit de tekst verduidelijkt. Zaken die aan bod kunnen komen in de inleiding (**Pollefliet2011**):

- context, achtergrond
- afbakenen van het onderwerp
- verantwoording van het onderwerp, methodologie
- probleemstelling
- onderzoeksdoelstelling
- onderzoeksvraag
- ...

1.1 Probleemstelling

Uit je probleemstelling moet duidelijk zijn dat je onderzoek een meerwaarde heeft voor een concrete doelgroep. De doelgroep moet goed gedefinieerd en afgelijnd zijn. Doelgroepen als "bedrijven," "KMO's," systeembeheerders, enz. zijn nog te vaag. Als je een lijstje kan maken van de personen/organisaties die een meerwaarde zullen vinden in deze bachelorproef (dit is eigenlijk je steekproefkader), dan is dat een indicatie dat de doelgroep goed gedefinieerd is. Dit kan een enkel bedrijf zijn of zelfs één persoon (je co-promotor/opdrachtgever).

1.2 Onderzoeksvraag

Wees zo concreet mogelijk bij het formuleren van je onderzoeksvraag. Een onderzoeksvraag is trouwens iets waar nog niemand op dit moment een antwoord heeft (voor zover je kan nagaan). Het opzoeken van bestaande informatie (bv. “welke tools bestaan er voor deze toepassing?”) is dus geen onderzoeksvraag. Je kan de onderzoeksvraag verder specifiëren in deelvragen. Bv. als je onderzoek gaat over performantiemetingen, dan

1.3 Onderzoeksdoelstelling

Wat is het beoogde resultaat van je bachelorproef? Wat zijn de criteria voor succes? Beschrijf die zo concreet mogelijk. Gaat het bv. om een proof-of-concept, een prototype, een verslag met aanbevelingen, een vergelijkende studie, enz.

1.4 Opzet van deze bachelorproef

De rest van deze bachelorproef is als volgt opgebouwd:

In Hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie.

In Hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen.

In Hoofdstuk 4, tenslotte, wordt de conclusie gegeven en een antwoord geformuleerd op de onderzoeksvragen. Daarbij wordt ook een aanzet gegeven voor toekomstig onderzoek binnen dit domein.

2. Stand van zaken

2.1 Angular

Angular is een framework dat het gemakkelijk maakt om webapplicaties te bouwen. Angular combineert templates, dependency injectie, end to end tooling en geïntegreerde best practices om hindernissen in de ontwikkeling op te lossen. Angular stelt ontwikkelaars in staat om applicaties te maken voor op het web, mobiel of desktop. (Lotanna, 2019)

Angular kan gezien worden als een volwaardig MVC[1] framework. Het wordt zo beschouwd omdat Angular de structuur van de applicatie in ontwikkeling sterk wil regulieren. Angular voorziet standaard veel functionaliteit. Aan de andere kant zorgt dit wel voor minder flexibiliteit. Angular verstrekt onderstaande functionaliteit. (Hamedani, 2018)

- Templates
- XSS[2] bescherming
- Dependency Injectie
- Component CSS encapsulatie
- Hulpmiddelen voor het unit-testen van componenten
- Ajax requests
- Routering
- Formulieren

Google, de ontwikkelaar van AngularJS, kondigde eind 2014 aan dat Angular 2 een volledige herschrijving van AngularJS zou zijn. Ze zouden zelf een nieuwe programmeertaal `AtScript` ontwikkelen die speciaal voor Angular 2 bedoeld was. Toen begon Microsoft

[1] Model View Controller
[2] Cross-Site Scripting

decorators te ondersteunen in hun taal TypeScript, waardoor deze de aanbevolen taal werd voor de ontwikkeling van Angular 2 applicaties. Het is echter ook mogelijk om applicaties te ontwikkelen in JavaScript (ECMAScript 5 en ECMAScript 6) en in Dart. (Fain, 2016)

Het team van Google besliste om geen grote updates meer te doen aan AngularJS omdat ze, naast het verbeteren van features en prestaties, Angular meer future-proof wilden maken door gebruik te maken van web componenten en ECMAScript 6. Het resultaat was Angular 2, verrijkt met enorm veel features. De voornaamste nieuwe features waren niet mogelijk in AngularJS waardoor een herschrijving nodig was. Als eerste was de ideologie veranderd. Bij AngularJS lag de focus op databinding en templates. Het belangrijkste doel was om af te komen van het traditionele proces voor DOM-handling via JavaScript's jQuery. Daarnaast moest het framework ook klaar voor de toekomst zijn. Sinds de meeste browsers ECMAScript 6 gebruiken, zullen applicaties dus ook ontwikkeld moeten worden in ES6. Angular 2 biedt die kans aan, met 100% browser achterwaartse compatibiliteit. Ook maakt Angular 2 gebruik van web componenten. Dit zijn JavaScript en HTML modules die gemaakt worden voor een specifieke taak in de applicatie. Als laatste werden routers, dependency injection, dynamic loading en async templating verbeterd of toegevoegd. Angular 2 werd uitgegeven in 2016. (Shan, 2015)

2.1.1 Inleiding

De basis-bouwstenen van Angular applicaties zijn NgModules. Deze voorzien compilatie context voor componenten en verzamelen gerelateerde code in functionele sets. Een Angular applicatie wordt bepaald door een set van NgModules. Elke applicatie heeft een root module dat bootstrapping mogelijk maakt. Daarnaast kunnen nog meerdere feature modules toegevoegd worden, maar deze zijn niet verplicht. (Angular, 2019a)

Componenten definiëren views, dit zijn sets van schermelementen die Angular kan gebruiken en aanpassen aan de hand van de logica in de applicatie. Daarnaast kunnen componenten services gebruiken die specifieke functionaliteiten voorziet die niet verbonden zijn met views. Deze services kunnen geïnjecteerd worden in componenten met behulp van dependency injection. Hierdoor wordt de code herbruikbaar en modulair. (Angular, 2019a)

Zowel componenten als services zijn gewone klassen met decorators. Deze kenmerken hun type en voorzien metadata. Angular gebruikt deze metadata om te weten hoe deze klassen gebruikt kunnen worden. De metadata van een component verbindt de component met een template die een view definieert. Een template combineert HTML met Angular directives en binding markup die Angular toestaan om de HTML aan te passen voordat het getoond wordt op het scherm. Daarnaast voorziet de metadata van een service de informatie die Angular nodig heeft om het beschikbaar te maken voor componenten via dependency injection. (Angular, 2019a)

2.1.2 Modules

Angular applicaties zijn `modulair`. NgModules zijn containers die samenhangende code van een bepaald domein of bepaalde workflow bevat. Ze kunnen componenten, services en andere code-bestanden die behoren tot de scope van de module bevatten. Ook kan functionaliteit geïmporteerd worden van een andere NgModule. Daarnaast kan een NgModule bepaalde functionaliteiten exporteren die andere NgModules kunnen gebruiken. Elke applicatie heeft minstens een NgModule klasse, de root module. Deze heet gewoonlijk AppModule en bevind zich in een bestand genaamd `app.module.ts`. De applicatie wordt gestart door deze root module te bootstrappen. Zoals eerder vermeld kunnen dus extra modules toegevoegd worden als feature modules. De AppModule kan deze extra modules omvatten in een hiërarchie met onbepaalde diepte. (Angular, 2019d)

Een NgModule wordt gedefinieerd door een klasse met de `@NgModule()` decorator. Deze decorator is een functie die een metadata object verwacht. De properties van dit object worden omschreven in de module. Enkele van de belangrijkste properties van een module zijn als volgt (Angular, 2019d):

- Declarations: set van componenten, directives en pipes die tot de module behoren.
- Exports: set van declarations die zichtbaar en bruikbaar moeten zijn in component templates van andere modules.
- Imports: set van modules die nodig zijn in component templates die gedefinieerd zijn in de huidige module.
- Providers: set van services (meer uitleg hier)
- Bootstrap: Dit is de view van de main applicatie, genaamd de root component. Deze host alle andere views van de applicatie. Dit kan enkel voor de root NgModule.

Modules voorzien compilatie context voor de componenten die ze declareren. De root module heeft altijd een root component die aangemaakt wordt tijdens de bootstrap. Elke module kan extra componenten omvatten die met behulp van de router geladen kunnen worden of aangemaakt kunnen worden door de template. (Angular, 2019d)

Een view wordt gedefinieerd door een component en zijn template. Een component kan een view hiërarchie bevatten die toestaat om een complexe samenstelling van een scherm te definiëren. Deze samenstelling kan als een eenheid aangemaakt, aangepast en verwijderd worden. De hiërarchie bestaat uit een enkele host view. Deze view kan de root van een hiërarchie zijn die `embedded views` kan bevatten, die op hun beurt host view zijn van een andere component. Deze componenten kunnen deel zijn van dezelfde module of geïmporteerd worden van een andere module. (Angular, 2019d)

De NgModules van Angular zijn totaal verschillend van de JavaScript modules. In JavaScript is elk bestand een module en elk object gedefinieerd in dat bestand behoort tot die module. Die objecten kunnen geëxporteerd worden door de module door gebruik te maken van het sleutelwoord `export`. Andere modules kunnen dan gebruik maken van dit publiek object door het sleutelwoord `import` te gebruiken. (Angular, 2019d)

2.1.3 Componenten

Elke applicatie heeft naast een module ook minstens een component, de root component. Die legt de verbinding tussen een component's view hiërarchie en het DOM. Elke component definieert een klasse die applicatiedata en -logica bevat. Deze wordt ook verbonden met een HTML-template die een view definieert. De `@Component()` decorator identificieert de onderstaande klasse als een component en voorziet de template en alle specifieke metadata voor die component. Deze metadata is net zoals bij NgModules een functie die een object verwacht. (Angular, 2019a)

Een component is dus de meest basis bouwsteen van een Angular applicatie. Zo'n applicatie bestaat meestal uit een `tree` van Angular componenten. Componenten zijn een subset van `directives` die altijd geassocieerd worden met een template. Echter in tegenstelling tot `directives` kan slechts een component geïnstantieerd worden per template-element. Daarnaast kan het gedrag van een component tijdens runtime ook geprogrammeerd worden door gebruik te maken van `lifecycle hooks` (**REFERENCE**)

De voornaamste properties van het metadata-object voor componenten zijn als volgt (Angular, 2019b)

- ChangeDetection: De change-detection strategie voor de component. Wanneer een component geïnstantieerd wordt, maakt Angular een change detector aan die verantwoordelijk is voor het uitdragen van de bindingen van de component. Er zijn 2 mogelijkheden nl. ChangeDetection#OnPush: CheckOnce(on demand) en ChangeDetection#Default: CheckAlways
- ViewProviders: definieert een set van injecteerbare objecten die zichtbaar zijn voor de DOM kinderen van de component's view.
- Template: Een inline template voor een Angular component.
- TemplateUrl: Het relatieve of absolute pad naar een template bestand voor een Angular component.
- Styles: Een of meerdere inline CSS stylesheets die gebruikt kunnen worden in de component
- StyleUrls: Een of meerdere relatieve of absolute paden naar bestanden die een CSS stylesheet bevatten om te gebruiken in de component.
- EntryComponents: een set van componenten die samen gecompileerd moeten worden met de component. Voor elke component hier gedefinieerd creëert Angular een `ComponentFactory` en slaat deze op in de `ComponentFactoryResolver`

Daarnaast erft het ook volgende properties over van de `@Directive()` decorator (Angular, 2019b)

- Selector: de CSS selector dat de directive identificieert in een template en de instantiatie van de directive start.
- Inputs: set van gegevensgebonden properties van de directive.
- Outputs: set van eventgebonden properties.
- Providers: configureert de injector van de directive met een token dat gemapt wordt naar een provider van een dependency

Angular componenten zijn een subset van directives die altijd verbonden zijn met een template. Anders dan andere directives kan maar een component geïnstantieerd worden per element in een template. Een component moet tot een NgModule behoren zodat die beschikbaar zou zijn voor andere componenten. (Angular, 2019a)

Een template combineert HTML met Angular markup die HTML-elementen kan aanpassen nog voordat deze getoond worden op het scherm. (Angular, 2019a)

Template directives voorzien logica en binding markup verbindt applicatie data met het DOM. Er zijn twee soorten data bindingen. (Angular, 2019a)

- `Event binding` staat toe om de applicatie te laten reageren op gebruikers invoer in de omgeving door het aanpassen van applicatie data.
- `Property binding` staat toe om berekende waarden uit de applicatie uit te schrijven in de HTML.

Voordat een view getoond wordt op een scherm zal Angular eerst de directives evalueren en de syntax van de binding in de template uitzoeken zodat de HTML elementen en het DOM aangepast kunnen worden, afhankelijk van de applicatie data en logica. Angular staat ook `two-way-binding` toe. Dit wil zeggen dat zowel aanpassingen in het DOM zoals gebruikers invoer de applicatie data aanpassen en applicatieberekeningen ook weergegeven zullen worden in de HTML. (Angular, 2019a)

Een template kan gebruik maken van `pipes` om de UX[3] te verbeteren door waarden te transformeren voor weergave. Angular voorziet enkele voorgedefinieerde pipes voor standaard transformaties, maar pipes kunnen ook zelf gedefinieerd worden met een eigen transformatie. (Angular, 2019a)

2.1.4 Services en DI

Voor data of logica die niet tot een specifieke view behoort en over de hele applicatie voorzien moet worden, kan een service klasse gebruikt worden. Een service wordt gedefinieerd met de `@Injecatble()` decorator. (Angular, 2019a)

Deze decorator voorziet de metadata die de klasse beschikbaar maakt voor creatie aan een `Injector` zodat deze geïnjecteerd kan worden als een dependency. Voor elke dependency moet een provider voorzien worden zodat de injector van deze provider gebruik kan maken om een dependency op te halen of een nieuwe aan te maken. Voor een service is dit standaard de service klasse zelf. Angular creëert een globale injector tijdens het bootstrap proces die gebruikt kan worden over de hele applicatie. Daarnaast kunnen extra injectors aangemaakt worden wanneer dat nodig is. (Angular, 2019e)

Dependency Injection (DI) is een belangrijk applicatie design pattern. Angular bevat een eigen DI framework dat kenmerkend gebruikt wordt in het design van Angular applicaties om op een efficiënte en modulaire manier om te gaan met het creëren van klassen. Een

[3] User Experience: het gevoel en de ervaring dat een gebruiker opdoet

klasse verzoekt dependencies aan een externe bronnen in plaats van deze zelf aan te maken. In Angular voorziet het DI framework de dependencies aan een klasse zodra deze geïnstantieerd is. (Angular, 2019c)

Services die aangemaakt worden via het Angular CLI commando `ng generate service [servicenaam]` registreert een provider voor de service in de root injector door `'root'` toe te kennen aan het `providedIn` property van het metadata object. Angular creëert dan een enkele instantie die geïnjecteerd wordt in elke klasse die daar om vraagt. Het `providedIn` property kan ook een specifiek type zijn. Hierdoor kan Angular de applicatie optimaliseren door de service te verwijderen uit de gecompileerde app wanneer dat type niet gebruikt wordt. Services kunnen ook geregistreerd worden in een specifieke NgModule. De service zal dan beschikbaar zijn voor alle componenten van die module. Om dit te bekomen, moet de service aan de `providers` set in de module toegevoegd worden. Als laatste kan een provider voor een service ook op component level geregistreerd worden. Op deze manier wordt een nieuwe instantie van de service gemaakt bij elke nieuwe instantie van de component. Dit kan door deze toe te voegen aan de `providers` property van het metadata object van de component. (Angular, 2019e)

Een component kan gebruik maken van een service of andere dependencies door deze te injecteren in de component met behulp van de `Injector`. Angular creëert alle injectors zelf, dus er moeten geen injectors aangemaakt of geïmplementeerd worden. Een injector maakt de dependencies aan en houdt een container van deze instanties bij zodat deze hergebruikt kunnen worden als dat mogelijk is. Op deze manier krijgt de component toegang tot de service of kan deze de dependency gebruiken. (Angular, 2019e)

Wanneer een nieuwe instantie van een component aangemaakt wordt, bepaalt Angular welke dependencies de component nodig heeft door te kijken naar de types van de parameters in de constructor. Dan checkt de injector de container met bestaande instanties. Als deze de gevraagde instantie vindt, wordt die teruggegeven. Indien dat niet het geval is, wordt een nieuwe instantie aangemaakt van de dependency en wordt deze opgeslagen in de container en teruggegeven. (**REFERENCE**)

2.1.5 Samenwerking

Al deze items vormen de basis over de hoofdzakelijke bouwstenen van een Angular applicatie. De afbeelding xx toont hoe deze bouwstenen met elkaar verbonden zijn en samenwerken.

Een component en een template vormen samen een Angular view. Een `@Component()` decorator voegt de metadata toe, waarbij een pointer zit naar de geassocieerde template. Directives en binding markup in een template kunnen de view aanpassen afhankelijk van applicatie data en logica. De dependency injector kan services in een component voorzien. Afhankelijk van de provider van de service wordt deze opnieuw aangemaakt. (Angular, 2019a)

2.2 React

React, aan de andere kant, is een bibliotheek met als doel ontwikkelaars te helpen met het bouwen van User Interfaces. De structuur van deze UI's worden voorgesteld als een boom waarbij de knopen componenten zijn. Een component bestaat uit zowel HTML als JavaScript die de logica beschrijft om deze te tonen. (Baer, 2018)

React is ontstaan binnen de Facebook Ads Org. Het team van Facebook bouwde standaard MVC client-side applicaties met two-way-binding en templates. In deze apps luisteren de views naar aanpassingen in de models en reageren op deze changes door zichzelf manueel up te daten. Deze applicaties waren in het begin heel simpel, maar naarmate het team van Facebook meer en meer features toevoegde, werden ze meer en meer complex. Om bij te blijven met deze complexe applicaties, werden ook meer en meer mensen bij het team toegevoegd, met als resultaat dat de applicaties moeilijk te onderhouden werden. Ze kwamen erachter dat in deze complexe apps een kleine aanpassing kon gebeuren, die ervoor zorgde dat de app moest uitzoeken welke views aangepast moesten worden en moest deze dan ook veranderen. Binnen het team noemden ze dit `cascading updates`. Deze updates gebeuren heel traag, want de app moest eerst en vooral bijhouden wat aangepast moest worden en wat niet en daarna de views zichzelf laten updaten. Het werd moeilijk om te voorspellen wat zorgde voor deze cascading updates. (Occhino, 2015)

De code die beschrijft hoe de applicatie eruit ziet, bestaat al. Het idee was dat telkens de data veranderde, enkel die code opnieuw uitgevoerd zou worden. Dit lijkt een goede oplossing, maar het probleem is dat op deze manier opgeslagen state verloren gaat bij de update en dus voor enkele glitches kon zorgen. Daarnaast was er ook nog een relatief groot verlies in tijd en een grote toename in CPU-gebruik op de client. Om dit op te lossen bedacht Jordan Walke een prototype die dit proces efficiënter maakt en een deftige `user experience` garandeert. Dit markeerde de geboorte van React. (Occhino, 2015)

Facebook.com maakte in 2012 volop gebruik van deze nieuwe technologie. Op dat moment werd het gebruikt om onder andere reacties te plaatsen op berichten, berichten leuk te vinden[4] en gesprekken te voeren in de chat. Toen Facebook Instagram overnam, kwam de vraag vanuit het Instagram team om React te gebruiken. Aangezien Facebook dit enkel intern gebruikte, vonden ze dat het lastig zou zijn om zaken los te koppelen van hun infrastructuur. Instagram wou absoluut deze technologie gebruiken en dat zorgde bij Facebook voor het initiatief om de versie van React aan te passen en opnieuw te bouwen zodat React `open sourced` kon worden. Op die manier zou Instagram dan ook gebruik kunnen maken van React. (Occhino, 2013)

2.2.1 JSX

JSX is een uitbreiding op de syntax van JavaScript. React raadt aan om JSX te gebruiken om te beschrijven hoe de UI er zou moeten uitzien. Het doet misschien herinneren aan een template taal, maar het komt samen met de kracht van JavaScript. JSX produceert React

[4] Toen bestond de optie nog niet om met emoticons berichten leuk te vinden.

elementen. (React, 2019f)

2.2.1.1 Redenen voor JSX

Bij React beseften ze dat de logica voor de weergave gekoppeld hangt met andere UI logica, zoals hoe events afgehandeld worden, hoe state verandert na verloop van tijd, en hoe data voorbereid wordt om weergegeven te worden. In plaats van technologieën te scheiden door markup en logica in aparte files te plaatsen, past React `seperation of concerns` toe door gebruik te maken van componenten die zowel markup als logica bevatten. Het is niet verplicht om JSX te gebruiken binnen React, maar het vormt een visueel hulpmiddel bij het werken aan de UI in de JavaScript code. Het staat ook toe om React meer handige en nuttige error- en waarschuwingsberichten te tonen. (React, 2019f)

2.2.1.2 Expressies insluiten in JSX

Elke geldige JavaScript expressie kan in JSX geplaatst worden door gebruik te maken van accolades. (React, 2019f)

2.2.1.3 JSX is ook een expressie

Na de compilatie worden JSX expressies gewone JavaScript functie calls. Dit betekent dat JSX gebruikt kan worden binnen `if` statements en in `for` loops. Daarnaast kan het ook toegekend worden aan variabelen, kan het aanvaard worden in argumenten en kan het geretourneerd worden van functies. (React, 2019f)

2.2.1.4 Attributen specifiëren met JSX

Binnen JSX kunnen aanhalingstekens gebruikt worden om string literals als attributen op te geven. Daarnaast kunnen ook accolades gebruikt worden om JavaScript expressies in te sluiten in een attribuut. Aanhalingstekens rond accolades zouden niet gebruikt mogen worden. Voor string waarden moeten aanhalingstekens gebruikt worden, voor expressies accolades. (React, 2019f)

Omdat JSX dichter aansluit bij JavaScript dan HTML, gebruikt React DOM de `camelCase` als naamgeving conventie in plaats van HTML attribuut namen. Bijvoorbeeld, `tabindex` in HTML wordt `tabIndex` in JSX. (React, 2019f)

2.2.1.5 JSX voorkomt injectie aanvallen

Het is volkomen veilig om gebruiker input te gebruiken in JSX. React DOM verzekert dat er geen data in de applicatie geïnjecteerd kan worden die niet bedoeld is voor die applicatie. Alle input wordt omgezet naar string voordat het weergegeven wordt. Dit helpt bij het voorkomen van XSS[5]. (React, 2019f)

[5] Cross Site Scripting

2.2.1.6 JSX stelt objecten voor

Babel zorgt ervoor dat JSX gecompileerd wordt naar `React.createElement()` calls. Deze call voert enkele controles uit die helpen om bugvrije code te schrijven. In principe creëert het een object genaamd `React element`. React leest deze objecten en gebruikt ze om het DOM op te stellen en up-to-date te houden. (React, 2019f)

2.2.2 Elementen weergeven

Elementen zijn de kleinste bouwstenen van React applicaties. Een element beschrijft wat getoond moet worden op het scherm. In tegenstelling tot browser DOM elementen zijn React elementen plain objects en dus gemakkelijk aan te maken. React DOM zorgt ervoor dat het DOM geüpdatet wordt zodanig dat het overeenkomt met deze elementen. (React, 2019i)

2.2.2.1 Elementen weergeven in het DOM

Applicaties binnen React hebben vaak een enkele root DOM node. Wanneer React geïntegreerd wordt in een bestaande applicatie, kunnen er oneindig veel geïsoleerde root DOM nodes zijn. Om een element in een root DOM node weer te geven, moeten beide meegegeven worden in `ReactDOM.render()`. (React, 2019i)

2.2.2.2 Een weergegeven element updaten

React elementen zijn onveranderlijk. Eens een element gecreëerd is, kunnen zijn kinderen noch zijn attributen aangepast worden. De enige manier om de UI aan te passen is om een nieuw element aan te maken en die door te geven aan de `ReactDOM.render()` functie. Echter de meeste applicaties roepen `ReactDOM.render()` slechts een keer op. In latere secties zal getoond worden hoe de code ingekapseld wordt in `stateful components`. (React, 2019i)

2.2.2.3 React updatet enkel de nodige zaken

React DOM vergelijkt het element en zijn kinderen met de vorige versie en laat enkel DOM updates doorgaan die nodig zijn om het DOM naar de gewenste staat te brengen. Enkel de nodes waarvan de inhoud aangepast werd, zal geüpdatet worden in het DOM. (React, 2019i)

2.2.3 Componenten en props

Componenten staan toe om de UI in onafhankelijke en herbruikbare delen op te splitsen. Componenten zijn net zoals JavaScript functies. Ze accepteren willekeurige inputs, genaamd `props` en retourneren React elementen die beschrijven wat op het scherm zou moeten komen. (React, 2019i)

2.2.3.1 Functie en klasse componenten

De gemakkelijkste manier om een React component te schrijven is door een JavaScript functie te schrijven. Deze componenten worden `functie componenten` genoemd omdat het ook letterlijk JavaScript functies zijn. Een andere manier is door gebruik te maken van het `ES6 class` keyword. In React's standpunt zijn beide manieren gelijkwaardig. Echter de `klasse componenten` hebben enkele extra karakteristieken. (React, 2019i)

2.2.3.2 Een component weergeven

React elementen kunnen naast DOM tags ook componenten voorstellen. Wanneer React een element tegenkomt dat een component voorstelt, geeft het de JSX attributen mee aan de component als een enkel object. Dit object wordt `props` genoemd. (React, 2019i)

Het is aangeraden om alle componenten te starten met een hoofdletter. React zal componenten die starten met een kleine letter behandelen als een DOM tag. (React, 2019i)

2.2.3.3 Componenten samenstellen

Componenten kunnen verwijzen naar andere componenten in hun uitvoer. Dit zorgt voor dezelfde abstractie van een component op gelijk welk niveau. In de meeste gevallen hebben React applicaties een `App` component helemaal bovenaan in de tree. Echter wanneer een bestaande applicatie React integreert, is het misschien beter om te beginnen met een kleine component aan te passen en geleidelijk naar boven in de view hiërarchie te werken. (React, 2019i)

2.2.3.4 Componenten extraheren

Sommige componenten kunnen opgedeeld worden in meerdere kleinere componenten. Het lijkt misschien veel werk, maar kan soms handig zijn om meer herbruikbare delen te hebben, zeker in grote applicaties. Een goede vuistregel is dat wanneer een deel van de UI meerdere keren gebruikt wordt of wanneer het complex genoeg is, dan is het een goede kandidaat om opgesplitst te worden in herbruikbare componenten. (React, 2019i)

2.2.3.5 Read-only props

Of een component gedeclareerd is als functie of als klasse, het mag nooit zijn eigen props aanpassen. Alle React componenten moeten zich als pure functies gedragen ten opzichte van hun props. Aangezien de meeste applicatie UI's dynamisch zijn en veranderen na verloop van tijd werd een concept ingevoerd, genaamd `state`. State staat React componenten toe om hun inputs aan te passen als reactie op invoer van de gebruiker en responses van het netwerk onder andere. Hierbij wordt de regel van `read-only props` niet overschreden. (React, 2019i)

2.2.4 State en levenscyclus

`State` is gelijkaardig aan `props`, maar het is private en volledig beheerd door de component zelf. (React, 2019a)

2.2.4.1 Lokale state toevoegen aan een klasse

Om state toe te voegen aan een klasse component, moeten enkele stappen overlopen worden. Als eerst moet er een klasse constructor toegevoegd worden die de initiële `this.state` toekent. Hierbij wordt `props` meegegeven aan de basis constructor. (React, 2019a)

setState()

`setState()` haalt wijzigingen in de state van de component op en geeft aan dat deze component en zijn kinderen opnieuw weergegeven moeten worden met de bijgewerkte state. Dit is de voornaamste methode die gebruikt wordt wanneer er updates moeten gebeuren in de UI als gevolg van event handlers en server responses. (React, 2019j)

De setState methode is eerder een request dan een bevel. Om betere prestaties te verkrijgen, kan React de update wat vertragen en dan later meerdere componenten in een keer updaten. React kan niet garanderen dat een wijziging in een component direct toegepast wordt. (React, 2019j)

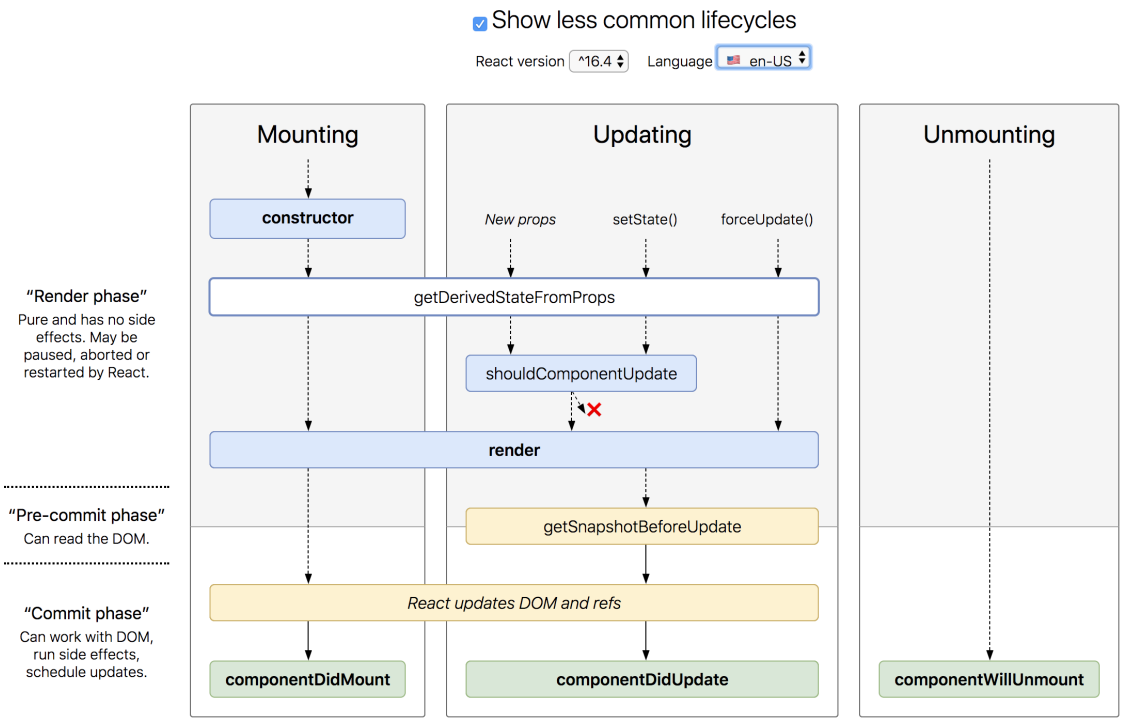
Omdat `setState()` niet altijd direct toegepast wordt, is het mogelijk dat het lezen van de state direct na de methode foutieve waarden teruggeeft. Om dit te voorkomen kan de `componentDidUpdate()` lifecycle methode of een `setState callback` gebruikt worden. (React, 2019j)

forceUpdate()

Normaal gezien zal een component opnieuw weergegeven worden wanneer zijn props of state aangepast wordt. Echter als de `render()` functie afhangt van bepaalde data geforceerd worden om een nieuwe weergave te laten doorgaan door de functie `forceUpdate()` te gebruiken. Door deze methode op te roepen zal `render()` opgeroepen worden op de component en de `shouldComponentUpdate()` wordt overgeslagen. Dit zorgt ervoor dat de normale lifecycle methodes in de kinderen getriggerd worden, waaronder ook de `shouldComponentUpdate()`. React zal nog steeds enkel updaten daar waar wijzigingen doorgebracht zijn. Het is aangeraden om `forceUpdate()` zoveel mogelijk te vermijden en enkel de props en state te lezen in de `render()` functie. (React, 2019j)

2.2.4.2 Lifecycle methodes toevoegen aan een klasse

In applicaties met vele componenten is het vaak belangrijk om resources zo snel mogelijk vrij te maken die door de componenten werden ingenomen als ze verwijderd worden. Er bestaan enkele speciale methodes binnen de component klasse die code uitvoeren wan-



Figuur 2.1: Dit diagram stelt een visuele referentie voor van de lifecycle methodes van een React component. Screenshot genomen op http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

neer de component gekoppeld of ontkoppeld wordt. Deze methodes worden `lifecycle methods` genoemd. Raadpleeg 2.1 voor een visuele referentie. (React, 2019a)

render()

De `render()` functie is de enige verplichte methode in een klasse component. Wanneer het opgeroepen wordt, moet het `this.props` of `this.state` overlopen en een van volgende types teruggeven: (React, 2019j)

- React elementen
- Arrays en fragmenten
- Portals
- Strings en numbers
- Booleans of null

De render functie zou puur moeten zijn, het mag de state van een component niet aanpassen. Elke keer dat het opgeroepen wordt, moet het hetzelfde resultaat retourneren en het heeft geen directe interactie met de browser. Wanneer er toch interactie met de browser moet zijn, kan er gebruik gemaakt worden van andere lifecycle methodes. `render()` zal niet uitgevoerd worden wanneer `shouldComponentUpdate()` `false` retourneert. (React, 2019j)

constructor()

Wanneer geen state geïnitialiseerd wordt en methodes niet gebonden worden, is er geen nood aan een contructor functie binnen de React component. De constructor van een component wordt aangeroepen voordat het gekoppeld wordt. Bij het implementeren van een constructor in een `React.component` subklasse moet `super(props)` opgeroepen worden als eerste statement. Als dat niet gebeurt, is `this.props` `undefined` en dat kan tot enkele rare bugs leiden. (React, 2019j)

In React bestaan constructor functies voor twee redenen. Als eerste om lokale state te initialiseren door een object toe te kennen aan `this.state`. Als tweede om event handler methodes te koppelen aan instanties. Binnen `constructor()` zou de call `setState()` nooit gebruikt mogen worden. In plaats daarvan kan de state direct geïnitialiseerd worden in de constructor. Dit is de enige plaats waar `this.state` direct toegewezen zou mogen worden. In alle andere methodes moet `this.setState()` gebruikt moeten worden. (React, 2019j)

componentDidMount()

Deze methode wordt opgeroepen direct nadat het gekoppeld werd. Alle initialisatie betreffende DOM nodes zou in `componentDidMount()` moeten komen. Als er data moet geladen worden vanuit een backend, dan is dit een goede plaats om een netwerk request te instantiëren. Het is wel belangrijk om deze subscriptions te sluiten in `componentWillUnmount()`. (React, 2019j)

In `componentDidMount()` kan de functie `setState()` direct gebruikt worden. Alhoewel dit voor een extra weergave zorgt, zal de gebruiker hiervan niets zien omdat dit gebeurt voordat de browser het scherm zal updaten. Dit is echter niet altijd aan te raden omdat dit vaak voor prestatieproblemen zorgt. In de meeste gevallen kan de initiële state in `constructor()` geïnitialiseerd worden, maar soms is het nodig om een DOM node af te meten voordat iets weergegeven wordt wanneer dit afhangt van de grote of plaats van de node. (React, 2019j)

componentDidUpdate()

`componentDidUpdate()` is aangeroepen direct nadat een update voorgekomen is. Deze methode wordt niet uitgevoerd na de initiële weergave. Deze methode zou gebruikt moeten worden om aan het DOM te werken nadat er een update gebeurd is. Het is ook een goede plaats om hierin netwerkverzoeken te doen zolang de huidige props met de vorige vergeleken worden. Een netwerk request is mogelijks niet nodig wanneer de props niet aangepast zijn. (React, 2019j)

Binnen de methode kan `setState()` opgeroepen worden, maar het moet gewrapt worden in een conditie. Wanneer dit niet gebeurt, zal dit een oneindige loop veroorzaken. `componentDidUpdate()` zal ook niet uitgevoerd worden wanneer `shouldComponentUpdate()` `false` teruggeeft. (React, 2019j)

componentWillUnmount()

Deze methode wordt aangeroepen onmiddellijk voordat een component ontkoppeld en vernietigd zal worden. Alle nodige clean-up zoals het annuleren van netwerk request en het opruimen van subscriptions zou in deze methode uitgevoerd moeten worden. (React, 2019j)

`setState()` zou nooit mogen opgeroepen worden in `componentWillUnmount()` omdat de component nooit meer opnieuw weergegeven zal worden. Eens een component losgekoppeld wordt, kan het nooit meer terug gekoppeld worden. (React, 2019j)

Zelden gebruikte lifecycle methodes

Volgende methodes kunnen in ongewone gevallen voorkomen. Ze kunnen soms handig zijn, maar in de meeste gevallen zijn ze niet nodig. (React, 2019j)

- shouldComponentUpdate()
- static getDerivedStateFromProps()
- static getDerivedStateFromProps()
- Error boundaries
- static getDerivedStateFromError()
- componentDidCatch()

2.2.4.3 State correct gebruiken

Wijzig state niet rechtstreeks

Wanneer state rechtstreeks aangepast wordt, zal de component niet opnieuw weergegeven worden. In plaats daarvan moet de methode `setState()` gebruikt worden. De enige plaats waar state direct toegewezen kan worden is in de constructor. (React, 2019j)

State updates kunnen asynchroon zijn

React kan meerdere `setState()` calls groeperen in een enkele update voor een betere prestatie. Omdat `this.props` en `this.state` asynchroon geüpdatet kunnen worden, is het niet aangeraden om te vertrouwen op die waarden om de volgende state te berekenen. Om te voorkomen dat de verkeerde waarden gebruikt worden in een `setState()` berekening, kan een tweede vorm van `setState()` gebruikt worden. Deze vorm verwacht een functie in plaats van een object. De functie ontvangt de vorige staat als eerste argument en de props van wanneer de update uitgevoerd wordt als tweede argument. (React, 2019j)

State updates zijn samengevoegd

Wanneer `setState()` aangeroepen wordt, zal React het meegegeven object samenvoegen met de bestaande state. Het samenvoegen is ondiep, dit betekent dat wanneer slechts een state attribuut moet veranderen, dan zal React enkel dat attribuut in de state aanpassen en

de rest onaangetast laten. (React, 2019j)

2.2.4.4 De data vloeit naar beneden

Zowel parent als kind kunnen niet weten of een bepaalde component stateful of stateless is en het zou ze niet mogen schelen of het gedefinieerd is als klasse of als functie. Daarom wordt state vaak lokaal of ingekapseld genoemd. Het is niet toegankelijk voor alle componenten dan degene die het bezit en kan aanpassen. Het is mogelijk voor een component om zijn state als props mee te geven aan child componenten. Dit wordt een `top-down` of een `unidirectionele` gegevensstroom genoemd. Een state is eigendom van een specifieke component en alle data of UI's die afgeleid zijn van die state kan enkel invloed hebben op componenten onder hen in de DOM tree. (React, 2019j)

2.2.5 Events afhandelen

Events afhandelen met React elementen is gelijkaardig aan het afhandelen van events in DOM elementen. Er zijn echter enkele verschillen in syntax. In de naamgeving van React events wordt gebruik gemaakt van `camelCase` in plaats van alles in `lowercase`. Daarnaast wordt met JSX een functie meegegeven als event handler in plaats van een string. Een ander verschil is dat men in React geen `false` kan retourneren om een event niet te laten doorgaan. Daarvoor moet expliciet de `preventDefault` methode gebruikt worden. (React, 2019e)

Met React zou normaal gezien geen `addEventListener` opgeroepen moeten worden om listeners toe te voegen op een DOM element. In plaats daarvan kan gewoon een listener voorzien worden bij de initiële weergave. (React, 2019e)

De betekenis van `this` is in JSX callbacks belangrijk. Net zoals in JavaScript zijn klasse methodes standaard niet gebonden. Wanneer een niet-gebonden methode meegegeven wordt aan een event handler, zal `this` in de callback `undefined` zijn. Dit is niet door React, maar door hoe JavaScript werkt. In het algemeen moeten methodes gebonden worden wanneer naar de methode gerefereerd wordt zonder haakjes. Binden gebeurt op de volgende manier: `this.method1 = this.method1.bind(this)`. (React, 2019e)

Als de class field syntax niet gebruikt wordt, kunnen ook `arrow functions` gehanteerd worden in de callback. Het probleem met die syntax is dat er telkens de component weergegeven wordt, er een nieuwe callback wordt gemaakt. In de meeste gevallen is dit geen probleem, maar als de callback meegegeven wordt aan een child component, dan moet die component een extra weergave doen. Daarom is het aangeraden om binding te doen in de constructor of class field syntax te gebruiken om dit soort prestatieverlies te voorkomen. (React, 2019e)

2.2.5.1 Argumenten meegeven aan event handlers

Binnen een loop is het niet ongewoon om een extra parameter te willen meegeven aan een event handler. In een `arrow function` moeten de argumenten expliciet meegegeven worden, terwijl met een `bind` worden alle argumenten automatisch doorgestuurd. (React, 2019e)

2.2.6 Conditionele weergave

In React kunnen afzonderlijke componenten gemaakt worden die het gedrag bevatten die de applicatie exact nodig heeft. Daarna kunnen enkele weergegeven worden, afhankelijk van de staat van de applicatie. Conditioneel weergeven werkt in React op dezelfde manier als condities werken in JavaScript. Door JavaScript operatoren zoals `if` of de `conditionele operator` toe te passen, kunnen elementen gecreëerd worden die de huidige staat van de applicatie voorstellen waarop React de UI zal updaten. (React, 2019c)

2.2.6.1 Elementen als variabelen

In React kunnen elementen opgeslagen worden in variabelen. Dit kan helpen om een deel van een component conditioneel weer te geven zonder dat de rest van de output gewijzigd wordt. Gebaseerd op de huidige staat kunnen dan bepaalde delen getoond worden. (React, 2019c)

Terwijl een variabele declareren en een `if` statement gebruiken een goede manier is om een component conditioneel weer te geven, is het soms beter om een kortere syntax te hanteren. In JSX zijn er bepaalde manieren om dit inline te doen. (React, 2019c)

2.2.6.2 Inline 'if' met de logische '&&' operator

In JSX kan gelijk welke expressie geïntegreerd worden door ze te wrappen in accolades. Dit geldt ook voor de logische `&&` operator van JavaScript. Dit kan handig zijn om conditioneel een element toe te voegen. Dit werkt omdat JavaScript enkel en alleen maar de tweede expressie in een `&&` operator zal evalueren wanneer de eerste `true` is. Als de eerste expressie `true` is, zal het element net na de `&&` in de output verschijnen, maar wanneer de eerste expressie `false` is, dan zal het element genegeerd worden en dus niet verschijnen. (React, 2019c)

2.2.6.3 Inline 'if-else' met de conditionele operator

Een andere manier om elementen conditioneel weer te geven, is door de conditionele operator `conditie ? true : false` van JavaScript te gebruiken. Deze manier kan gebruikt worden voor grotere expressies, maar in dat geval wordt het vaak minder leesbaar. Net als in JavaScript is het aan de developer om een gepaste stijl te kiezen die leesbaar is voor zichzelf en eventueel het team. Wanneer een conditie te ingewikkeld wordt, is het misschien handig om componenten te extraheren zoals besproken in 2.2.3.4. (React,

2019c)

2.2.6.4 Voorkomen dat componenten weergegeven worden

In heel uitzonderlijke gevallen is het gewenst om component te verbergen, ook al werd het weergegeven door een andere component. Dit kan door `null` te retourneren in plaats van zijn render output. `Null` retourneren in de render methode van een component beïnvloedt niets in de levenscyclus van die component. Alle lifecycle methodes zullen nog steeds aangeroepen worden. (React, 2019c)

2.2.7 Lijsten en sleutels

In JavaScript kan een lijst getransformeerd worden de `map()` functie toe te passen. De functie overloopt alle elementen in de lijst, waarop dan een behandeling wordt toegepast. De functie op zich retourneert een nieuwe lijst. In React kan een array getransformeerd worden in een lijst van elementen op ongeveer dezelfde manier. (React, 2019h)

2.2.7.1 Meerdere componenten weergeven

In React kunnen collecties van elementen gecreëerd worden en in JSX opgenomen worden door accolades te gebruiken. Door elk element in een array te mappen naar een DOM object en het resultaat op te slaan in een variabele, kunnen meerdere componenten weergegeven worden door enkel het resultaat weer te geven. (React, 2019h)

2.2.7.2 Een basische lijst component

Lijsten worden bij voorkeur weergegeven in een component. Echter wanneer een component een array binnenkrijgt via `props`, wordt er een waarschuwing gegenereerd dat een `key` moet voorzien worden voor de items in de lijst. Een `key` / `sleutel` is een speciaal string attribuut moet opgenomen worden bij het maken van lijsten. (React, 2019h)

2.2.7.3 Sleutels

`Sleutels` zijn een attribuut die React helpen om bij te houden welke items gewijzigd, toegevoegd of verwijderd werden. Elk element in een array zou een `key` moeten krijgen om het een stabiele identiteit te geven. De beste manier om een `key` te kiezen is om een string te gebruiken die een uniek item uit de lijst identificeert onder de andere items uit die lijst. In de meeste gevallen wordt het ID-attribuut gebruikt. Als er geen unieke ID's zijn, kan de index van het item gebruikt worden, maar best als laatste redmiddel. Indexen als `keys` gebruiken kan enkel en alleen maar als de volgorde van de items niet verandert. Anders kan dit effect hebben op de prestaties en negatieve gevolgen voor de state van de component. Wanneer geen expliciete `key` wordt voorzien, zal React standaard de index gebruiken. (React, 2019h)

2.2.7.4 Sleutels moeten enkel uniek zijn binnen de array

Sleutels die gebruikt worden binnen een array moeten enkel uniek zijn in die array. Ze moeten dus niet globaal uniek zijn. Dezelfde sleutels kunnen gebruikt worden voor verschillende arrays. Sleutels worden enkel gebruikt als hint voor React. Ze worden niet doorgegeven aan componenten. Als een component die waarde toch nodig heeft, kan die als een prop worden doorgegeven onder een andere naam. (React, 2019h)

2.2.7.5 Map() insluiten in JSX

JSX staat toe om gelijk welke expressie in te sluiten in accolades. Dit omvat dus ook het resultaat van een `map()` functie. Soms kan dit resulteren in code die beter leesbaar is, maar deze stijl kan ook misbruikt worden. Wanneer de body van de `map()` functie te genest wordt, dan is het waarschijnlijk een goed idee om de component te extraheren zoals besproken in 2.2.3.4. (React, 2019h)

2.2.8 Formulieren

Form elementen in HTML werken iets anders dan andere DOM elementen in React, omdat form elementen standaard een interne state bijhouden. Wanneer een standaard HTML formulier ingediend wordt, dan zal de gebruiker naar een nieuwe pagina geleid worden. In de meeste gevallen is dit niet de gewenste functionaliteit. Meestal is het handig om een JavaScript functie te hebben die het indienen van de form behandelt en daarbij toegang heeft tot de inhoud van de form die door de gebruiker werd ingevuld. De standaard manier om dit te bereiken in React is aan de hand van een techniek die `gecontroleerde componenten` wordt genoemd. (React, 2019d)

2.2.8.1 Gecontroleerde componenten

In HTML onderhouden elementen zoals `<input>`, `<textarea>` en `<select>` vaak hun eigen state en updaten het volgens de input van de user. In React wordt veranderlijke state bijgehouden door een property in de component. Beide manieren kunnen gecombineerd worden door de state in React de enige bron van waarheid te maken. De component die de form weergeeft, beheert ook wat er gebeurt wanneer de gebruiker iets wijzigt in de form. Een form element waarvan de waarde van de input op deze manier gecontroleerd wordt door React wordt een `gecontroleerde component` genoemd. (React, 2019d)

Wanneer de `setState()` wordt uitgevoerd na de `onChange()` event, kan de state van de React component beschouwd worden als de enige bron van waarheid. Met een `gecontroleerde component` heeft elke verandering een bijhorende handlerfunctie. Dit maakt het gemakkelijk om de input van gebruikers aan te passen of te valideren. (React, 2019d)

2.2.8.2 De textarea tag

In HTML wordt een `<textarea>` tag gedefinieerd door zijn kinderen. Echter, in React wordt een `value` attribuut gebruikt in plaats. Op deze manier kan een form die gebruik maakt van een `<textarea>` geschreven worden op een gelijkaardige manier als een `<input>` tag. (React, 2019d)

2.2.8.3 De select tag

In HTML zal een `<select>` tag een `drop down` lijst genereren. De items binnen deze lijst zijn alle `<option>` children binnen de `<select>` tag. Een initieel item kan geselecteerd worden door aan de `<option>` het `selected` attribuut te vermelden. In plaats van dergelijk attribuut te gebruiken, hanteert React een `value` attribuut op de root `<select>`. Voor een `gecontroleerde component` veel handiger omdat het maar een enkel item moet updaten. (React, 2019d)

2.2.8.4 De file input tag

In HTML kan een gebruiker met behulp van `<input type=''file'' />` een of meerdere files vanuit hun opslag uploaden naar een server of laten manipuleren door JavaScript via de `file API`. Omdat deze `value read-only` is, wordt dergelijk component in React een `uncontrolled component` genoemd. (React, 2019d)

2.2.8.5 Meerdere inputs behandelen

Bij het behandelen van meerdere input elementen in een form kan een `name` attribuut aan elk element toegevoegd worden, waarna een handler functie beslist wat te doen gebaseerd op de `event.target.name`. (React, 2019d)

2.2.8.6 Alternatieven voor gecontroleerde componenten

Soms kan het wel veel werk zijn om gecontroleerde componenten te maken, omdat voor elke manier waarop data aangepast kan worden een event handler geschreven moet worden en alle state moet door een React component gesluisd worden. Dit kan vooral vervelend worden wanneer een bestaande applicatie omgezet moet worden naar een React applicatie. In deze situatie is het misschien handig om gebruik te maken van `ongecontroleerde componenten`. (React, 2019d)

2.2.9 State naar boven verheffen

Vaak moeten meerdere componenten dezelfde veranderlijke state afspiegelen. In dat geval wordt aangeraden om de state te verheffen naar de dichtste gelijke voorouder. Dit wordt `lifting state up` genoemd. De lokale state wordt verplaatst naar een voorouder en van daaruit doorgegeven en aangepast. De voorouder wordt dan de `bron van waarheid`.

Aangezien alle componenten die data nodig heeft van deze bron een (klein)kind is van die component, zal de data altijd in sync zijn. (React, 2019g)

In een React applicatie zou er voor elke data die verandert slechts een `bron van waarheid` mogen zijn. Normaal gezien wordt state eerst toegevoegd aan de component die het nodig heeft om weergegeven te worden. Daarna kan het, als andere componenten het ook nodig hebben, naar boven verheven worden naar de dichtstbijzijnde gemeenschappelijke voorouder. In plaats van state te proberen synchroniseren, zou vertrouwd moeten worden op een `top-down data flow`. (React, 2019g)

Het verheffen van state houdt in dat er meer code geschreven moet worden, maar als voordeel heeft het dat het minder werk zal inhouden om bugs te vinden en isoleren. Omdat state zich maar in slechts een enkele component bevindt, zijn de locatiemogelijkheden van de bug veel kleiner geworden. (React, 2019g)

2.2.10 Compositie versus overerving

React heeft een krachtig compositiemodel en raadt aan om compositie te gebruiken om code tussen componenten opnieuw te gebruiken in plaats van overerving. Bij het gebruik van overerving komen vaak problemen voor die opgelost kunnen worden met compositie. (React, 2019b)

2.2.10.1 Insluiting

Sommige componenten weten nog niet op voorhand wat hun kinderen zullen zijn. Bij zulke componenten is het aangeraden om de speciale `children` prop te gebruiken om de children direct in hun output te plaatsen. Op die manier kunnen andere componenten willekeurige children meegeven aan die component door ze te nesten in JSX. (React, 2019b)

2.2.10.2 Specialisatie

In sommige gevallen kunnen bepaalde componenten gezien worden als een speciaal geval van een generieke component. Zo kan bijvoorbeeld een `WelcomeDialog` een speciale `Dialog` component zijn. In React wordt dit ook bereikt via compositie, waar een specifieke component een meer generieke component weergeeft en het configureert met props. (React, 2019b)

2.2.10.3 Wat met overerving?

Bij Facebook gebruiken ze React met duizenden componenten en ze hebben, naar eigen zeggen, nog geen enkele use case tegengekomen waarbij ze overerving zouden aanraden. Props en compositie geven de nodige flexibiliteit om het uiterlijk en gedrag van een component aan te passen. (React, 2019b)

2.3 Mithril.js

Mithril is een modern client-side framework, gemaakt voor het bouwen van single page applications. Het is klein in download, snel in performance en voorziet routing en XHR-tools. (Mithril, 2019c)

Dit framework is vergelijkbaar met React, maar het is eenvoudiger en meer capabel. Mithril vervangt de behoefte aan bibliotheken zoals jQuery. De kleine bestandsgrootte en gemakkelijke API zorgt ervoor dat Mithril ideaal is om geïntegreerde JavaScript-widgets of user interfaces die hoge performance vereisten hebben, te maken. (Gilbert, 2018)

Mithril was oorspronkelijk geschreven door Leo Horie. Het framework werd uitgebreid en verbeterd tot wat het tegenwoordig is dankzij het werk van de community. (Mithril, 2019b)

2.3.1 Virtuele DOM nodes

2.3.1.1 Virtueel DOM

Een virtuele DOM-structuur is een JavaScript-datastructuur die een `DOM tree` beschrijft. Het bestaat uit een aantal geneste virtuele DOM nodes, genaamd `vnodes` (Mithril, 2019g)

Meestal worden virtuele DOM's opnieuw opgebouwd bij een rendercyclus. Deze cyclus vindt plaats na een event of na data changes. Bij zo'n rendercyclus bekijkt Mithril de vorige versie van de vnode-structuur en wijzigt enkel de DOM-elementen op plaatsen waar een aanpassing plaatsvond. Nieuwe vnodes maken is goedkoper dan het wijzigen van de DOM. (Mithril, 2019g)

2.3.1.2 Basis

Vnodes zijn JavaScript objecten die delen van het DOM voorstellen. De virtuele DOM-machine van Mithril verbruikt de vnodes-structuur en stelt een DOM-structuur op. Vnodes kunnen aangemaakt worden aan de hand van het `m()` hyperschrift. Hyperschrift kan ook `componenten` verbruiken. (Mithril, 2019g)

2.3.1.3 Structuur

Vnodes zijn JavaScript objecten die een element voorstellen. Ze kunnen volgende eigenschappen hebben:

(Mithril, 2019g)

2.3.1.4 Soorten vnodes

In Mithril bestaan er 5 verschillende soorten vnodes. Deze wordt bepaald door de `tag` attribuut.

tag	De `nodeName` van een DOM element. De tag kan ook een string zijn als de vnode een fragment, een text-node of vertrouwd HTML node is.
key	De waarde die gebruikt wordt om een DOM element te mappen naar zijn respectievelijke item in een array van data.
attrs	Een hashmap van DOM attributen, events, eigenschappen en lifecycle methodes.
children	Een array van de kinderen van de vnode. Meestal zijn dit ook vnodes, maar in sommige gevallen kan dit een string, een number of een boolean zijn.
text	Dit wordt enkel gebruikt als een vnode enkel een text node heeft als kind. Dit wordt enkel gedaan om performance redenen.[6]
dom	Verwijst naar het element dat correspondeert met de vnode.
domSize	Deze eigenschap wordt enkel gebruikt in fragmenten en vertrouwde HTML nodes. Het stelt het aantal DOM elementen voor die de vnode vertegenwoordigt.
state	Een object dat voorzien wordt door de `core engine` en bewaard wordt doorheen cyclussen. Bij een POJO component vnode erft het state object de eigenschappen van de component klasse/object over.
events	Een object dat de event handlers opslaat zodat deze later verwijderd kunnen worden door gebruik te maken van de DOM API. Net zoals het state-object wordt deze ook bewaard doorheen de rendercyclussen. Deze eigenschap wordt intern gebruikt door Mithril en zou nooit gebruikt of aangepast mogen worden.
instance	Een object voor componenten. Het is een opslaglocatie voor de waarde die de view methode teruggeeft. Deze eigenschap wordt ook intern gebruikt door Mithril en zou nooit gebruikt of aangepast mogen worden.

Element	Vertegenwoordigt een DOM element.
Fragment	Vertegenwoordigt een lijst van DOM elementen waarvan de parent ook andere DOM elementen kan bevatten die zicht niet in de lijst bevinden.
Text	Vertegenwoordigt een DOM text-node
Trusted HTML	Vertegenwoordigt een lijst van DOM elementen van een HTML string
Component	Vertegenwoordigt het DOM dat gegenereerd wordt bij het weergeven van de component. Dit kan enkel als de `tag` een component is die een `view` methode bevat.

(Mithril, 2019g)

In een virtuele DOM structuur moet alles een vnode zijn, dus ook text. De `m()` utility normaliseert zijn `children` argument en verandert tekst in text vnodes en geneste arrays in fragment vnodes. Het eerste argument van de `m()` functie kan enkel een element tag naam of een component zijn. Trusted HTML vnodes kunnen gemaakt worden aan de hand van de methode `m.trust()` (Mithril, 2019g)

2.3.1.5 Monomorfe klasse

De `mithril/render/vnode` wordt gebruikt door Mithril om alle vnodes aan te maken. Dit zorgt ervoor dat moderne JavaScript machines optimaal virtuele DOM structuren kunnen vergelijken door altijd vnodes te compileren met deze ene verborgen klasse. (Mithril, 2019g)

2.3.1.6 Anti-patterns vermijden

Vnodes vertegenwoordigen de staat van het DOM op een gegeven tijdstip. De rendermachine van Mithril gaat ervan uit dat een vnode die opnieuw gebruikt wordt onaangepast is. Het aanpassen van een vnode die gebruikt werd in een vorige weergave zal resulteren in een `undefined` gedrag. (Mithril, 2019g)

Het is mogelijk om vnodes te hergebruiken om een render cycle te vermijden, maar het is beter om gebruik te maken van de `onbeforeupdate` hook te gebruiken. (Mithril, 2019g)

2.3.2 Componenten

2.3.2.1 Structuur

Componenten zijn een mechanisme die gebruikt worden om delen van de view in te kapselen. Hierdoor wordt de code gemakkelijker om te structureren of om opnieuw te gebruiken. Elk JavaScript object dat een `view` methode heeft is een Mithril component. Deze componenten kunnen gebruikt worden door gebruik te maken van de `m()` functie. (Mithril, 2019a)

2.3.2.2 Lifecycle methodes

Componenten hebben dezelfde lifecycle methodes als vnodes. Bij elke lifecycle methode wordt een `vnode` als argument meegegeven, net zoals bij de `view` methode. Zoals de andere types vnodes kunnen componenten extra lifecycle methodes hebben wanneer ze gebruikt worden als vnode type. (Mithril, 2019a)

Lifecycle methodes in vnodes overschrijven de component methodes niet, ook niet omgekeerd. De lifecycle methodes van componenten worden altijd opgeroepen na de methodes van de vnode. Dezelfde naam kiezen voor callback functies als de lifecycle methode namen

zou dus altijd vermeden moeten worden. (Mithril, 2019a)

2.3.2.3 Data meegeven aan componenten

Om data mee te geven aan componenten kan door een `attrs` object mee te geven als tweede parameter in de hyperschrift functie. Deze data kan worden geraadpleegd in de view van de component of in de lifecycle methodes via `vnode.attrs` (Mithril, 2019a)

2.3.2.4 State

Zoals alle virtuele DOM nodes kunnen component vnodes een state hebben. Op deze manier kunnen zaken als objectgeoriënteerde structuren, inkapseling en separation of concerns ondersteund worden. (Mithril, 2019a)

Een component aanpassen zorgt niet voor een DOM update. Dit gebeurt wanneer een event handler getriggerd wordt, wanneer een HTTP request (gemaakt via `m.request`) vervuld is of wanneer de browser navigeert naar een andere route. Om een redraw geforceerd te laten doorgaan kan gebruik gemaakt worden van `m.redraw()`. (Mithril, 2019a)

Closure component state

Het is mogelijk om component state te gebruiken bij POJO componenten, maar dat is in het algemeen niet de beste aanpak. Een betere manier is door gebruik te maken van een closure component. Dat is simpelweg een wrapper functie die een POJO component instantie retourneert die op zich zijn eigen closed-over scope meedraagt. (Mithril, 2019a)

Met deze closure componenten kan state bewaard worden door variabelen die gedeclareerd staan in de buitenste functie. Ook functies die in de closure component gedeclareerd staan hebben toegang tot de state variabelen. (Mithril, 2019a)

Deze closure componenten worden op dezelfde manier gebruikt als POJO componenten. Het voordeel hiervan is dat er geen gebruik gemaakt moet worden van `this`. (Mithril, 2019a)

POJO component state

Het is altijd aangeraden om closure componenten te gebruiken wanneer state moet bijgehouden worden. Er kan echter toch state bijgehouden worden in POJO componenten en die kan dan op drie manieren opgevraagd worden, namelijk als blueprint bij initialisatie, via `vnode.state` of via het `this` keyword in component methodes. (Mithril, 2019a)

2.3.2.5 Klassen

Componenten kunnen ook geschreven worden aan de hand van het `class` sleutelwoord. Deze class componenten moeten een `view()` methode definiëren. Deze methode wordt

gedetecteerd via `.prototype.view` om in de DOM structuur weergegeven te worden. Ook klasse componenten worden op dezelfde manier gebruikt als normale componenten. Ook in klasse componenten kan state bijgehouden worden en deze kan beheerd worden door zijn attributen en methodes. State wordt aangeroepen via het `this` sleutelwoord, maar om de juiste referentie te verkrijgen moeten arrow functions gebruikt worden voor de event handler callbacks. (Mithril, 2019a)

2.3.2.6 Gemixte soorten componenten

Componenten kunnen gelijk welke soort component als kind hebben. Er zijn geen restricties op de soort kinderen qua component. (Mithril, 2019a)

2.3.2.7 Speciale attributen

Mithril gebruikt enkele property keys. Het is aangeraden om deze te vermijden in/als attributen van normale componenten. Deze keys zijn (Mithril, 2019a)

- Lifecycle methodes, nl. `oninit`, `oncreate`, `onbeforeupdate`, `onupdate`, `onbeforeremove` en `onremove`
- `key`
- `tag`

2.3.2.8 Anti-patterns vermijden

Hoewel Mithril heel flexibel is, is het toch afgeraden om bepaalde patterns te vermijden. (Mithril, 2019a)

'Fat' componenten vermijden

In het algemeen is een 'fat' component een component met eigen instance methodes. Functies in `vnode.state` of in `this` zijn af te raden omdat deze niet hergebruikt kan worden in andere componenten. Het is ook eenvoudiger om de code te refactoren wanneer deze in de data laag staat in plaats van in state. Code die hergebruikt moet worden, wordt best in een module geplaatst. (Mithril, 2019a)

vnode.attrs niet meegeven andere componenten

Om de interface flexibel te maken en een simpele implementatie te garanderen lijkt het logisch om alle attributen (`vnode.attrs`) door te geven aan de child componenten. In plaats daarvan zouden slechts enkele attributen meegegeven mogen worden aan child componenten. (Mithril, 2019a)

Vermijd het maken van component definities in views

Als er een component gecreëerd wordt binnen een view functie, dan zal elke redraw een andere clone hebben van de component. Dit betekent dat componenten die dynamisch aangemaakt werden via een factory ook altijd opnieuw van nul worden aangemaakt. (Mithril, 2019a)

Vermijd het maken van component instanties buiten views

Als een component instantie buiten een view functie gemaakt wordt, dan zullen toekomstige redraws de vnode vergelijken, maar omdat deze dezelfde referentie krijgt, zal deze vnode overgeslagen worden en er zal geen update gebeuren. (Mithril, 2019a)

2.3.3 Lifecycle methodes

2.3.3.1 Gebruik

Componenten en vnodes kunnen allemaal lifecycle methodes hebben, ook `hooks` genoemd. Deze methodes worden opgeroepen op verschillende tijdstippen tijdens het bestaan van een DOM element. (Mithril, 2019e)

Alle lifecycle methodes krijgen de vnode als eerste argument. Het `this` keyword in de callback van de methodes wordt gekoppeld aan de `vnode.state`. Hooks kunnen enkel aangeroepen worden als gevolg van de `m.render()` call. Ze worden niet aangeroepen als het DOM aangepast wordt zonder Mithril. (Mithril, 2019e)

2.3.3.2 De levenscyclus van een DOM element

Een DOM element wordt meestal gecreëerd en aan het document toegevoegd. Daarna kunnen zijn attributen of children geüpdatet worden wanneer een UI event getriggerd wordt. Het kan ook verwijderd worden van het document. (Mithril, 2019e)

Wanneer een element verwijderd wordt, kan het tijdelijk opgeslagen worden in een `memory pool`. Elementen in deze pool kunnen hergebruikt worden in een volgende update. Dat proces heet `DOM recycling`. Door het recycleren van een bestaand element kunnen de prestatiekosten van het maken van een nieuw element vermeden worden. (Mithril, 2019e)

Oninit

De `oninit(vnode)` hook wordt aangeroepen voordat een vnode behandeld wordt door de virtuele DOM machine. Deze hook wordt gegarandeerd uitgevoerd voordat zijn DOM element vastgemaakt wordt aan het document en voordat zijn kinderen uitgevoerd worden. In de oninit(vnode) hook zou de `vnode.dom` nooit opgeroepen mogen worden. Deze methode wordt niet aangeroepen wanneer een element geüpdatet wordt, maar wel wanneer een verwijderd element gerecycleerd wordt uit de memory pool. De oninit hook is vooral

nuttig voor het initialiseren van component state aan de hand van de argumenten die meegegeven werden met `vnode.attrs` of `vnode.children`. (Mithril, 2019e)

Oncreate

De `oncreate(vnode)` hook wordt aangeroepen nadat een DOM element gecreëerd is en toegevoegd is aan het document. Deze lifecycle methode wordt gegarandeerd uitgevoerd op het einde van de render cyclus. Net zoals oninit wordt deze hook ook niet aangeroepen wanneer een element geüpdatet wordt. Echter, anders dan oninit kunnen elementen waarvan de vnodes een oncreate hook hebben niet gerecycleerd worden. De oncreate hook is handig voor het lezen van lay-out waarden die een redraw kunnen veroorzaken, voor het starten van animaties en voor het initialiseren van derde partij bibliotheken die een referentie nodig hebben naar het DOM element. (Mithril, 2019e)

Onupdate

De `onupdate(vnode)` hook wordt aangeroepen nadat een DOM element geüpdatet werd terwijl het aan het document verbonden is. De methode wordt gegarandeerd uitgevoerd op het einde van de render cyclus. Deze hook wordt enkel aangeroepen als het element al bestond in de vorige render cyclus. Het wordt nooit aangeroepen wanneer het element gecreëerd of gerecycleerd wordt. DOM elementen waarvan zijn vnodes een onupdate hook hebben kunnen niet gerecycleerd worden. De onupdate hook is vooral handig voor het lezen van lay-out waarden die een redraw kunnen veroorzaken en voor het dynamisch aanpassen van state in derde partij bibliotheken die de UI[7] aanpassen, nadat model data gewijzigd is. (Mithril, 2019e)

Onbeforeremove

De `onbeforeremove(vnode)` hook wordt aangeroepen voordat een DOM element van het document losgekoppeld wordt. Wanneer een promise geretourneerd wordt, zal Mithril het element slechts loskoppelen wanneer de promise voldaan is. Deze hook wordt enkel aangeroepen op de elementen die hun `parentNode` verliezen, maar het wordt niet aangeroepen in zijn kinderen. Ook hier worden elementen waarvan de vnodes een onbeforeremove hook bevatten, niet gerecycleerd. (Mithril, 2019e)

Onremove

De `onremove(vnode)` hook wordt aangeroepen voordat een DOM element verwijderd wordt van het document. Deze methode wordt uitgevoerd op elk element dat uit het document verwijderd wordt, ongeacht of het rechtstreeks van de parent werd losgemaakt of dat het een child is van een ander element dat werd losgemaakt. De onremove hook is nuttig voor het runnen van opkuis taken. (Mithril, 2019e)

[7] User Interface

Onbeforeupdate

De `onbeforeupdate(vnode, old)` hook wordt aangeroepen voordat een vnode vergeleken wordt in een update. Als deze functie false retourneert, dan voorkomt Mithril dat in deze vnode een aanpassing gebeurt, waardoor ook de kinderen niet aangepast worden. Deze hook is handig voor het verminderen van de vertraging tijdens updates. (Mithril, 2019e)

2.3.4 Keys

Keys zijn een mechanisme die toelaten om DOM elementen te herschikken in een NodeList. Een key wordt gebruikt om DOM element te verbinden met een dataobject aan de hand van het id van dat object. Het key property zou altijd het unieke id-attribuut van objecten binnen een array moeten zijn. (Mithril, 2019d)

2.3.4.1 Gebruik

Een veelgebruikt patroon is om data te hebben die bestaat uit een array van objecten, hiervan wordt dan een lijst van vnodes gegenereerd die elk object in de array mappen. (Mithril, 2019d)

Het probleem hierbij ligt bij het aanpassen van die array. Stel dat uit een array van twee objecten het eerste item werd weggehaald, dan weet Mithril niet of het eerste object werd verwijderd of het tweede werd verwijderd en het eerste aangepast werd. Dit kan voor problemen zorgen bij het focussen van elementen of wanneer er stateful jQuery plug-ins worden gebruikt, kunnen deze een foutieve interne state hebben na de update. (Mithril, 2019d)

Wanneer een lijst van vnodes afkomstig is uit een dynamische array van data is het beter om een `key` property toe te voegen aan elke vnode. Dit zorgt ervoor dat Mithril op een slimme manier het DOM kan herschikken zodat elk DOM element juist gemapt blijft aan zijn respectievelijke item in de data array. (Mithril, 2019d)

2.3.4.2 Single-child keyed fragmenten

Soms is het nodig om een component op bevel opnieuw te initialiseren. Hiervoor kan het patroon van een `single-child keyed fragment` gebruikt worden, waarbij de key aangepast wordt om het element te verwijderen en opnieuw te initialiseren. (Mithril, 2019d)

Het kan ook gebonden worden aan een gekende identiteit, bijvoorbeeld wanneer een backend wordt aangesproken om een item op te halen aan de hand van een id. Wanneer het id in de routing wordt gebruikt, kan de key gekoppeld worden aan `m.route.params('id')`. Wanneer de user niet meer naar die ene item wil kijken, zal de key veranderen, waardoor het element verwijderd zal worden van het document. (Mithril, 2019d)

2.3.4.3 Key gerelateerde problemen debuggen

Keys kunnen soms tot verwarrende problemen leiden wanneer het verkeerd gebruikt wordt. Een typisch symptoom is dat de applicatie state corrupt blijkt te zijn geworden na enkele gebruiker interactie, meestal wanneer data verwijderd wordt. (Mithril, 2019d)

Vermijd wrapper elementen rond keyed elementen

Keys moeten geplaats worden op de vnode dat een direct kind is van de array. Dat betekent dat wanneer een element gewrapt wordt in een div element, de key eigenlijk in het div element moet geplaats worden. (Mithril, 2019d)

Vermijd het verbergen van keys in component root elementen

Wanneer de code gerefactored wordt en een component gemaakt wordt, dan moet de key verdwijnen uit de component. In de m utility functie moet de key dan geplaatst worden op de component zelf. (Mithril, 2019d)

Vermijd het wrappen van keyed elementen in arrays

Arrays zijn vnodes en daarvoor ook keyable. Keyed elementen zouden nooit gewrapt mogen worden in arrays buiten single-child keyed fragmenten. (Mithril, 2019d)

Vermijd variabele types

Keys moeten altijd van het type string zijn. Als ze dat niet zijn, worden ze gecast naar een string. Om dezelfde keys te vermijden (bijvoorbeeld string ‘1‘ en number 1 zou in dezelfde key resulteren) zou maar een type gebruikt mogen worden. (Mithril, 2019d)

Vermijd het gebruik van keyed en niet-keyed vnodes in dezelfde array

Een array van vnodes kan enkel keyed vnodes of niet-keyed vnodes bevatten, niet beide. Wanneer dit toch gebeurt, zal er een error optreden. `null`, `undefined` of booleans worden beschouwd als niet-keyed vnodes. (Mithril, 2019d)

Vermijd het doorgeven van model data direct naar de component wanneer het model key gebruikt als data property

Het is mogelijk dat er een `key` attribuut nodig is in het data model. Dit kan voor problemen zorgen in combinatie met Mithril’s `key` logica. Om problemen te voorkomen zou de data gewrapt moeten worden zodat Mithril deze niet beschouwt als render instructies. (Mithril, 2019d)

2.3.5 Het auto-redraw systeem

Mithril implementeert een virtueel DOM vergelijkingssysteem om snel te kunnen weergeven. Daarbij biedt het ook meerdere mechanismes om gedetailleerde controle te krijgen over de weergave van een applicatie. (Mithril, 2019f)

Het auto-redraw systeem van Mithril synchroniseert het DOM wanneer er aanpassingen gemaakt zijn in de data laag. Het systeem wordt ingeschakeld wanneer `m.mount` of `m.route` aangeroepen wordt, maar blijft uitgeschakeld als de applicatie uitsluitend gebootstrapt wordt door `m.render` calls. Wanneer er toch een redraw zou moeten gebeuren, kan deze manueel getriggerd worden door gebruik te maken van `m.redraw()`. Deze methode zal een asynchrone redraw starten. (Mithril, 2019f)

Het auto-redraw systeem bestaat simpelweg uit een re-render functie die wordt uitgevoerd op de achtergrond als bepaalde functies voltooid zijn. (Mithril, 2019f)

2.3.5.1 Na event handlers

Mithril zal automatisch een redraw laten doorgaan na DOM event handlers die gedefinieerd staan in een Mithril view. Deze redraw kan manueel uitgeschakeld worden door gebruik te maken van de instelling `e.redraw` op `false` te zetten. (Mithril, 2019f)

2.3.5.2 Na m.request

Mithril zal ook automatisch een redraw laten doorgaan wanneer een `m.request` voltooid wordt. De redraw kan gedeactiveerd worden door een extra object toe te voegen aan de request, waarbij het object een `background` property heeft die op true staat. (Mithril, 2019f)

2.3.5.3 Na een verandering in route

Ook na een `m.route.set()` call en na een routewijziging via links (door gebruik te maken van `m.route.Links`) zal Mithril een redraw laten doorgaan. (Mithril, 2019f)

3. Methodologie

3.1 Overzicht

3.1.1 Installatie en indeling

3.1.1.1 Installatie

Het aanmaken van een nieuwe applicatie in Angular, React en Mithril verloopt in alle drie de gevallen tamelijk vlot. Hoewel er bij Mithril meer handelingen nodig zijn, verloopt het toch vlotter dan Angular en React die in slechts twee commands een applicatie kunnen maken. Figuur 3.1 toont de nodige commands om een applicatie aan te maken en te laten uitvoeren. Angular en React zijn al volledig geconfigureerd. Echter bij Mithril zijn er extra handelingen vereist voordat de applicatie uitgevoerd kan worden:

- In het `package.json` bestand moet er een `start` entry toegevoegd worden aan het `scripts` object. Hoe dit object er uit moet zien, kan gevonden worden in figuur 3.2.
- Er moet een `src/index.js` bestand aangemaakt worden. Om te starten met een ”Hello World” kan de code uit figuur 3.3 gebruikt worden.
- Als laatste moet er een `index.html` gecreëerd worden. Hoe de HTML er uit moet zien, kan gevonden worden in figuur 3.4.
- Daarna kan `index.html` geopend worden in de browser om het resultaat te kunnen bekijken.

```
> npm i -g @angular/cli
> ng new my-angular-app

> cd my-angular-app
> ng serve
```

(a) Angular setup

```
> npm i -g create-react-app
> npx create-react-app my-react-app

> cd my-react-app
> npm start
```

(b) React setup

```
> npm init --yes
> npm i mithril --save
> npm i webpack webpack/cli --save-dev

> npm start
```

(c) Mithril setup

Figuur 3.1: De nodige commands om een nieuwe applicatie op te zetten.

```
{} package.json > ...
  6   "scripts": {
  7     "start": "webpack src/index.js --output bin/app.js -d --watch"
  8   },
```

Figuur 3.2: De `start` entry in het `scripts` object binnen `package.json`.

3.1.1.2 Indeling

Nadat de applicaties aangemaakt zijn, kan gekeken worden naar de structuur die Angular, React en Mithril hanteren. Dit is belangrijk om het standpunt van alle drie de technologieën te achterhalen.

Bij het bekijken van de architectuur die gebruikt wordt in Angular applicaties, is het duidelijk dat Angular zijn folderstructuur baseert op de componenten binnen de applicatie. Binnen de `src` folder die Angular zelf aanmaakte zit er een `app` folder die een basis module en component bevat. De module bestaat uit een bestand `app.module.ts`, terwijl de component uit vier bestanden bestaat zoals getoond in figuur 3.5a. Deze bestanden zijn:

- app.component.css
- app.component.html
- app.component.spec.ts
- app.component.ts

React daarentegen heeft geen `out-of-the-box` structuur buiten de `src` folder. Zoals figuur 3.5b aantoont bevat die folder de volgende bestanden:

- App.css
- App.js
- App.test.js
- index.css

```
src > JS index.js
  1   import m from "mithril";
  2
  3   m.render(document.body, "hello world");
```

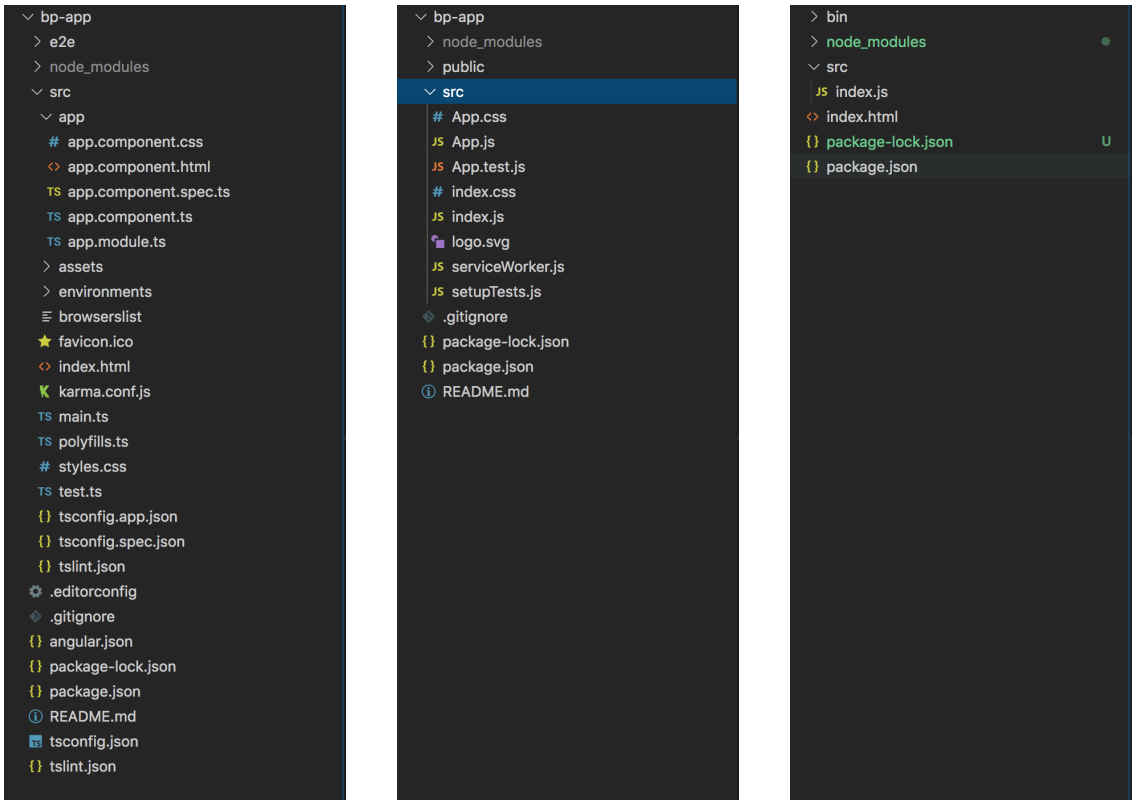
Figuur 3.3: De code nodig binnen het `src/index.js` bestand om 'hello world' weer te geven.

```
index.html > ...
1   <!DOCTYPE html>
2   <body>
3       <script src="bin/app.js"></script>
4   </body>
```

Figuur 3.4: De HTML binnen het `index.html` bestand.

- index.js
- logo.cvg
- serviceWorker.js
- setupTests.js

Het is duidelijk dat Mithril geen structuur bevat aangezien de `src` folder tijdens de setup van de applicatie zelf aangemaakt moet worden. Mithril's architectuur ziet er dan uit zoals figuur 3.5c aantoont.



(a) Angular (b) React (c) Mithril

Figuur 3.5: De architecturen na het opzetten van een nieuwe applicatie.

3.1.2 Grootte

De grootte van de bestanden in een applicatie is belangrijk. JavaScript code moet gedownloaded worden voordat de applicatie volledig functioneel is. Hoe meer code er is, hoe langer het zal duren om de applicatie uitgevoerd kan worden. Dat beïnvloedt de responsiviteit die de gebruiker waarneemt en dat is heel belangrijk. De downloads hebben ook invloed op het verbruik in bandbreedte. (Coulman, 2015)

De grootte van het onderliggende framework is niet het enige dat telt. Het is ook belangrijk om de grootte van de code van de applicatie in overweging te brengen. Een framework kan een grote invloed hebben op de grootte van de applicatiecode. Een framework met veel uitgewerkte features staat ontwikkelaars toe om minder eigen applicatiecode te schrijven. Bij het vergelijken van grootte is de totale grootte (framework + app code) de beslissende factor. (Coulman, 2015)

Om de grootte te kunnen vergelijken, willen we ook hetzelfde weergeven wanneer de applicatie uitgevoerd wordt. Daarom wordt om te beginnen in alle drie de applicaties een component aangemaakt die een `h1` element weergeeft met ”Hello World!” als inhoud. Figuur 3.6 schetst die situatie.

(a) Angular (b) React (c) Mithril

Figuur 3.6: De ”Hello World!” DOM-structuren nadat tekst werd weergegeven via componenten.

Wanneer dan gekeken wordt naar de grootte van de applicaties zoals berekend in figuur 3.7 is het duidelijk dat Angular het meeste geheugen inneemt. React is iets kleiner in geheugenbezetting, maar Mithril steekt er met kop en schouders bovenuit als kleinste van de groep. Tabel 3.1 geeft de benaderingen weer.

```
MBP-van-Jelle:bp-projects jelle$ du -sh *
357M    Angular-App
 61M    Mithril-App
245M    React-App
MBP-van-Jelle:bp-projects jelle$
```

Figuur 3.7: De grootte van de applicaties kan bekeken worden door het commando `du -sh *` uit te voeren.

	Grootte (MB)
Angular	357
React	245
Mithril	61

Tabel 3.1: Deze tabel geeft de benaderingen weer van de grootte van elke ‘’Hello World!” applicatie.

3.2 Functionaliteiten

3.2.1 Componenten en templates

In Angular, React en Mithril zijn componenten gewoon JavaScript of TypeScript klassen. Echter in React moet er een `render()` functie aanwezig zijn en in Mithril wordt een `view()` fucntie verplicht. Figuur 3.8 toont hoe de componenten er uitzien.

In Angular worden templates voorzien in een apart bestand, maar het is ook mogelijk om dit binnen het bestand van de component te doen aan de hand van het `template` attribuut voor het meta-data object van de `@Component` decorator. Figuur 3.8a toont een voorbeeld van dat `template` attribuut.

React voorziet geen apart bestand, alles qua template gebeurt in de speciale `render()` functie waarin JSX geplaatst kan worden zoals in figuur 3.8b.

Ook in Mithril is er geen aparte template file. Alles gebeurt binnen de return van de `view()` functie. Figuur 3.8c toont hoe dat in zijn werk gaat.

```angular
import { Component } from '@angular/core';

@Component({
  selector: 'app-home-component',
  template: '<h1>Hello World!</h1>',
  styleUrls: ['./home-component.component.css']
})
export class HomeComponentComponent {

}
```

(a) Angular

```react
import React, {Component} from 'react'

class HomeComponent extends Component {
    render() {
        return (
            <h1>Hello World!</h1>
        )
    }
}

export default HomeComponent;
```

(b) React

```mithril
import m from 'mithril'

class HomeComponent {
    view() {
        return m('h1', 'Hello World!');
    }
}

export default HomeComponent;
```

(c) Mithril

Figuur 3.8: Componenten met hun ”Hello World!” template.

4. Conclusie

A. Onderzoeksvoorstel

Het onderwerp van deze bachelorproef is gebaseerd op een onderzoeksvoorstel dat vooraf werd beoordeeld door de promotor. Dat voorstel is opgenomen in deze bijlage.

A.1 Introductie

JavaScript is een van de slechtste programmeertalen. De object prototypen zijn een primitieve en slordige manier om aan OO-programmeren te doen. Die zorgen ervoor dat het niet goed schaalt met grote applicaties. (Eng, 2016)

Technologieën zoals React of Angular hebben de laatste tijd wat interesse gewonnen. React wordt meer en meer gebruikt, maar Angular blijft toch wel populair binnen de business. In full-stack developer vacatures wordt Angular in 59% van de gevallen genoemd, React slechts in 37% (Schlothauer, 2019)

Dit werk heeft als bedoeling de keuze van technologie voor webapplicaties gemakkelijker te maken. Dit zal vooral voordelig zijn voor bedrijven binnen webdevelopment. De keuze van technologie kan bepalend zijn of een project al dan niet succesvol opgeleverd wordt. We stellen hier de vraag: "Welke eigenschappen van een project bepalen de aan te raden keuze van technologie voor dit project?". Ook zal bepaald worden bij welke eigenschappen een technologie als Angular of React beter is dan puur JavaScript.

A.2 State-of-the-art

JavaScript is altijd al een van de meest gebruikte programmeertalen geweest. JavaScript is een scripttaal die gebruikt wordt voor het interactief maken van een webpagina. De kenmerken van JavaScript worden omschreven als:

- Snel
- Gemakkelijk
- Krachtig

(Garbadge, 2018)

React is een JavaScript library die het voor de gebruiker gemakkelijk maakt om User Interfaces te maken. Het werd door Facebook ontwikkeld in 2011 met als doel de code voor grote webapplicaties gemakkelijker beheersbaar te maken. (Chand, 2019)

Angular is een herschrijving van het AngularJS framework ontwikkeld door Google. De eerste publieke versie (Angular2) werd vrijgegeven in september 2016. AngularJS begon populariteit te verliezen in het voordeel van de nieuwe Angular2. Op 28 mei 2019 werd versie 8 uitgerold. (Bodrov-Krukowski, 2018)

Er werden onderzoeken en online artikels gevonden die de verschillen tussen React en Angular benaderen, zoals Ari (2018). Ook werd al onderzoek gedaan naar het verschil met Vue.js, een JavaScript framework, maar er zijn geen onderzoeken gevonden over het verschil met pure JavaScript.

A.3 Methodologie

Eerst zullen er applicaties ontwikkeld worden met verschillende eigenschappen, zoals grootte, soorten data en complexiteit in JavaScript, Angular en React. Daarna zullen deze applicaties vergeleken worden op basis van bepaalde eigenschappen.

Snelheid van compilatie

De snelheid van compilatie is de eerste factor die onderzocht zal worden. De tijd zal gemeten worden en daarna wordt die per technologie vergeleken met elkaar.

Prestatie in runtime

Voor gebruikers van de webapplicatie is de prestatie in runtime heel belangrijk. Tegenwoordig zijn mensen gewoon van een snelle internetverbinding te hebben, dus moet een webapplicatie ook snel geladen worden. Er zal een backend opgesteld worden waaruit data opgevraagd zal worden. Deze backend is voor alle technologieën dezelfde. Er zal o.a. nagegaan worden hoe snel de data opgehaald kan worden.

Eenvoud

Om grote applicaties te onderhouden, is eenvoud een van de meest belangrijke aspecten. In dit onderzoek zal bepaald worden hoe gemakkelijk het is om binnen de gebruikte technologie een aanpassing of uitbreiding te maken. Ook zal er nagegaan worden hoeveel lijnen code nodig waren en welke handelingen uitgevoerd moesten worden om tot het eindresultaat te komen.

De eigenschappen van de webapplicaties zullen gemeten worden binnen een virtuele machine. De gebruikte software voor de verschillende applicaties in alle drie de technologieën zal Visual Studio Code van Microsoft zijn.

Op basis van de eigenschappen zal dan een overzicht opgesteld worden waarin de technologieën vergeleken zullen worden bij de verschillende applicaties.

A.4 Verwachte resultaten

Vermoedens zijn dat aan de hand van de resultaten Angular en React heel snel de voorkeur zullen krijgen op JavaScript. Vermoedelijk zullen Angular of React de code overzichtelijker maken, hoewel dat voor een langere compilatietijd zal zorgen. JavaScript zal echter wel nog enkele voordelen metzich meebrengen bij enkele soorten webapplicaties tegenover Angular of React.

A.5 Verwachte conclusies

De verwachtingen voor dit onderzoek zijn:

- JavaScript is sneller bij kleine en gemiddelde applicaties met weinig functionaliteit
- Voor webapplicaties met veel functionaliteit en verschillende soorten data zullen Angular en React overzichtelijker zijn.
- JavaScript is trager dan Angular of React bij applicaties met veel klassen of interfaces
- React zal de snelste zijn wanneer het komt op het aanpassen van het DOM.
- Bij grote en complexe applicaties zal JavaScript moeilijker te programmeren zijn
- De performance van JavaScript zal in elk van de gevallen beter zijn.

Bibliografie

Angular. (2019a). Architecture overview. Verkregen 20 mei 2019, van https://angular.io/guide/architecture

Angular. (2019b). Component. Verkregen 20 mei 2019, van https://angular.io/api/core/Component

Angular. (2019c). Dependency Injection in Angular. Verkregen 20 mei 2019, van https://angular.io/guide/dependency-injection

Angular. (2019d). Introduction to modules. Verkregen 20 mei 2019, van https://angular.io/guide/architecture-modules

Angular. (2019e). Introduction to services and dependency injection. Verkregen 20 mei 2019, van https://angular.io/guide/architecture-services

Ari, D. (2018, september 12). A comparison between angular and react and their core languages. Verkregen 3 april 2019, van https://medium.freecodecamp.org/a-comparison-between-angular-and-react-and-their-core-languages-9de52f485a76

Baer, E. (2018). *What is React and why it matters*. Sebastopol, California, United States: O'Reilly Media, Inc.

Bodrov-Krukowski, I. (2018, maart 22). Angular Introduction: What It Is, and Why You Should Use It. Verkregen 13 oktober 2019, van https://www.sitepoint.com/angular-introduction/

Chand, S. (2019, oktober 13). What Is React? – Unveil The Magic Of Interactive UI With React. Verkregen 13 oktober 2019, van https://www.edureka.co/blog/what-is-react/#3

Coulman, R. (2015, februari 17). JavaScript Framework Size. Verkregen 29 december 2019, van https://randycoulman.com/blog/2015/02/17/javascript-framework-size/

Eng, R. K. (2016, januari 26). The top 10 things wrong with JavaScript. Verkregen van https://medium.com/javascript-non-grata/the-top-10-things-wrong-with-javascript-58f440d6b3d8

Fain, Y. (2016, april 26). Angular 2 and TypeScript - A High Level Overview. Verkregen 20 mei 2019, van https://www.infoq.com/articles/Angular2-Typescript-High-Level-Overview

Garbadge, M. J. (2018, december 30). Top 3 most popular programming languages in 2018 (and their annual salaries). Verkregen van https://hackernoon.com/top-3-most-popular-programming-languages-in-2018-and-their-annual-salaries-51b4a7354e06

Gilbert. (2018, december 6). Mithril.js: A Tutorial Introduction (Part 1). Verkregen 2 december 2019, van https://gilbert.ghost.io/mithril-js-tutorial-1/

Hamedani, M. (2018, november 5). React vs. Angular: The Complete Comparison. Verkregen 3 april 2019, van https://programmingwithmosh.com/react/react-vs-angular/

Lotanna, N. (2019, maart 6). New Angular features you didn't know existed. Verkregen 1 december 2019, van https://blog.logrocket.com/new-angular-features-you-didnt-know-existed-7f292a6e7afc/

Mithril. (2019a). Components. Verkregen 6 december 2019, van https://mithril.js.org/components.html

Mithril. (2019b). Credits. Verkregen 6 december 2019, van https://mithril.js.org/credits.html

Mithril. (2019c). Introduction. Verkregen 2 december 2019, van https://mithril.js.org/

Mithril. (2019d). Keys. Verkregen 21 december 2019, van https://mithril.js.org/keys.html

Mithril. (2019e). Lifecycle methods. Verkregen 20 december 2019, van https://mithril.js.org/lifecycle-methods.html

Mithril. (2019f). The auto-redraw system. Verkregen 22 december 2019, van https://mithril.js.org/autoredraw.html

Mithril. (2019g). Virtual DOM nodes. Verkregen 4 december 2019, van https://mithril.js.org/vnodes.html

Occhino, T. (2013). JS Apps at Facebook (JSConfUS 2013). Verkregen 22 mei 2019, van https://www.youtube.com/watch?v=GW0j4sNH2w

Occhino, T. (2015, januari 28). Introducing React Native (React.js Config 2015). Verkregen 21 mei 2019, van https://www.youtube.com/watch?v=KVZ-PZI6W4

React. (2019a). Components and Props. Verkregen 22 december 2019, van https://reactjs.org/docs/components-and-props.html

React. (2019b). Composition vs Inheritance. Verkregen 23 december 2019, van https://reactjs.org/docs/composition-vs-inheritance.html

React. (2019c). Conditional rendering. Verkregen 23 december 2019, van https://reactjs.org/docs/conditional-rendering.html

React. (2019d). Forms. Verkregen 23 december 2019, van https://reactjs.org/docs/forms.html

React. (2019e). Handling events. Verkregen 22 december 2019, van https://reactjs.org/docs/handling-events.html

React. (2019f). Introducing JSX. Verkregen van https://reactjs.org/docs/introducing-jsx.html

React. (2019g). Lifting State Up. Verkregen 23 december 2019, van https://reactjs.org/docs/lifting-state-up.html

React. (2019h). Lists and Keys. Verkregen 23 december 2019, van https://reactjs.org/docs/lists-and-keys.html

React. (2019i). Rendering Elements. Verkregen van https://reactjs.org/docs/rendering-elements.html

React. (2019j). State and lifecycle. Verkregen 22 december 2019, van https://reactjs.org/docs/state-and-lifecycle.html

Schlothauer, S. (2019, maart 12). React steadily grows, while Angular maintains enterprise hold. Verkregen 18 september 2019, van https://jaxenter.com/react-angular-enterprise-156725.html

Shan, P. (2015, november 21). What's new in Angular 2.0? Why it's rewritten - addressing few confusions. Verkregen 20 mei 2019, van http://voidcanvas.com/angular-2-introduction