Proof of concept: De update automatiseren van Angular versie 16 naar versie 20 in de applicaties van een end-to-end kredietdienstverlener.

Wauters Sander.

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

Promotor: Mevr. I. Malfait
Co-promotor: Dhr. P. De Seranno
Academiejaar: 2025–2026
Eerste examenperiode

Departement IT en Digitale Innovatie .

HO
GENT

Woord vooraf

Deze bachelorproef vormt het sluitstuk van mijn opleiding Toegepaste Informatica, met als specialisatie Mobile en Enterprise Development. Tijdens deze opleiding kreeg ik de kans om aan verschillende projecten te werken en deed ik ervaring op met verschillende programmeertalen en frameworks. Wat mij daarbij het meest boeide, was het ontwikkelingsproces zelf. Om die reden koos ik ervoor om mijn bachelorproef uit te werken rond een technisch onderwerp.

De originele probleemstelling komt van mijn co-promotor, Peter De Seranno. Het probleem was dat er verschillende Angular-applicaties geüpdatet moesten worden naar een nieuwe versie. Wat mij opviel, was het grote aantal herhalingen in dit updateproces. Processen die zich herhalen zijn geschikt om te automatiseren. Zo ontstond het idee om software te ontwikkelen die het updateproces kan ondersteunen. Dit project was een leerrijke ervaring waarin ik mijn technische kennis kon toepassen. Verder has het Angular framework nieuw voor mij. Het was interessant om hier meer over te leren.

Voor dit project kon ik rekenen op de hulp en begeleiding van enkele personen die ik graag wil bedanken. In de eerste plaats wil ik mijn promotor, mevrouw Irina Malfait, bedanken voor de opvolging en begeleiding van mijn werk. Haar gerichte en constructieve feedback hielp mij om deze bachelorproef naar een hoger niveau te brengen. Daarnaast wil ik mijn oprechte dank uitspreken aan mijn co-promotor, Peter De Seranno, voor zijn technische ondersteuning en het nalezen van mijn werk. Zijn input zorgde ervoor dat ik het overzicht behield en dat het onderzoek steeds praktisch en relevant bleef.

Tot slot wil ik mijn ouders bedanken voor hun voortdurende steun en begrip. Dankzij hun aanmoediging kon ik de overstap maken naar deze studierichting en mijn opleiding met vertrouwen verderzetten. Met deze bachelorproef hoop ik iets bij te dragen aan het ontwikkelingsproces en de manier waarop we software refactoren en onderhouden.

Samenvatting

Het Angular framework vereenvoudigt het ontwikkelingsproces voor het bouwen van dynamische webapplicaties. Zoals bij de meeste software ontvangt Angular regelmatig updates. Deze updates zijn noodzakelijk, omdat ze de cyberveiligheid verbeteren. Het toepassen van dergelijke updates is echter niet altijd vanzelfsprekend. Angular verwijdert verouderde functionaliteiten uit het framework. Daardoor moet de broncode van Angular-applicaties aangepast worden. Dit type updates vindt om de 6 à 12 maanden plaats. Bij meerdere enterprise-applicaties kan de benodigde tijd voor dit updateproces snel oplopen. Het bedrijf Stater ervaart dit probleem. Stater is een end-to-end dienstverlener voor zowel hypothecaire als consumentenkredieten. Zij willen al hun applicaties updaten van Angular v16 naar v20. De sprong van vier versies betekent dat er vermoedelijk veel aanpassingen in de broncode nodig zijn.

Om deze reden wil dit onderzoek achterhalen in welke mate de automatisering van het updateproces van Angular v16 naar v20, over meerdere applicaties, de onderhoudstijd voor ontwikkelaars kan verlagen. Om hierop een antwoord te formuleren, ontwikkelt dit onderzoek een applicatie om het updateproces te ondersteunen. Deze applicatie, de *updater*, fungeert als proof of concept. Voor de implementatie van de updater maken we gebruik van zoek- en vervangfuncties in combinatie met de TypeScript Compiler API. Om de updater te ontwikkelen, voorzien we een collectie aan helperfuncties. De updater zelf is hard-coded. Dit geeft de nodige flexibiliteit om verschillende aanpassingen te automatiseren. Om de updater te gebruiken bij toekomstige updates, dient deze opnieuw ingesteld te worden.

Met onze aanpak is het mogelijk om een kwart van alle nodige wijzigingen in de update van v16 naar v20 te automatiseren. Daarnaast zijn we niet beperkt tot aanpassingen in Angular. Onze aanpak werkt op alle TypeScript code binnen een project. De updater is minder geschikt om wijzigingen aan logica uit te voeren. Aanpassingen die inzicht vereisen in de werking van de applicatie vormen een uitdaging. Om de updater optimaal te benutten, is het aangewezen deze te gebruiken bij de automatisatie van syntactische aanpassingen.

Inhoudsopgave

Lijst van figuren

Lijst van tabellen

Lijst van codefragmenten

1

Inleiding

Softwareframeworks vereenvoudigen het maken van dynamische webapplicaties. Het bedrijf Stater gebruikt voor deze toepassing het Angular framework. Net zoals de meeste software krijgt Angular geregeld updates. Deze updates zijn noodzakelijk, omdat ze de cyberveiligheid verbeteren. Daarnaast brengen deze updates verschillende voordelen, zoals: nieuwe functionaliteiten, betere performantie, bugfixes, ... Het toepassen van deze updates is niet altijd even vanzelfsprekend. Nieuwe functionaliteiten dienen soms als vervanging voor oudere. Het gevolg hiervan is dat de broncode van applicaties die Angular gebruiken moet veranderen.

Stater is een end-to-end dienstverlener voor zowel hypothecaire als consumentenkredieten. Ze verzorgt namens de kredietverstrekker de volledige dienstverlening aan de klanten van deze kredietverstrekkers. Stater heeft intern meerdere applicaties die gebruikmaken van het Angular framework. Specifiek gebruiken deze applicaties Angular versie 16 (v16). Stater zou graag al deze applicaties updaten naar de meest recente versie, Angular versie 20 (v20).

1.1. Probleemstelling

De sprong van 4 versies betekent dat er wellicht veel aanpassingen aan de broncode nodig zijn. Dit probleem groeit met de grootte van de broncode en verder met het aantal applicaties dat deze updates nodig heeft. Het manueel uitvoeren van al deze veranderingen neemt veel tijd in beslag. Tenslotte is dit geen eenmalig probleem. Angular krijgt volgens Callaghan (2023) een nieuwe versie om de 6 maanden.

Het tijdig uitvoeren van deze updates is in de praktijk niet altijd mogelijk. De ontwikkelaars hebben ook andere taken dan enkel software te onderhouden. Ondertussen worden nieuwe applicaties ontworpen of worden huidige applicaties uitgebreid. Uiteraard kan dit soort onderhoud niet eeuwig uitgesteld worden. Software-

updates zijn volgens Vaniea en Rashidi (2016) noodzakelijk om de cyberveiligheid van een applicatie te garanderen. De huidige versies van Angular zijn momenteel zonder veiligheidsrisico's, maar dit betekent niet dat er geen zijn. Er bestaat altijd een kans dat een nieuw risico gevonden wordt. Het is daarom van belang om alles regelmatig up-to-date te houden.

Dit alles tezamen zorgt ervoor dat de onderhoudskosten snel oplopen. De studie door U. Kaur en Singh (2015) beweert dat het onderhouden van een softwareproject gemiddeld 60% van de totale kostprijs in beslag neemt. Om deze redenen is het vereenvoudigen van het updateproces waardevol. Voor de programmeurs die de updates toepassen, vermindert de werkdruk. En voor het bedrijf betekent dit dat de onderhoudstijd/-kosten voor hun applicaties lager kunnen liggen.

1.2. Onderzoeksvraag

Op basis van de bovenstaande probleemstelling is de volgende onderzoeksvraag geformuleerd: in welke mate kan de automatisering van het updateproces van Angular v16 naar v20, bij meerdere applicaties, de onderhoudstijd voor de ontwikkelaars verlagen? Om deze onderzoeksvraag te beantwoorden, zijn de volgende deelvragen opgesteld:

- Welke veranderingen moeten uitgevoerd worden om Angular van v16 naar v20 te updaten?
- Welke manieren bestaan er om code automatisch aan te passen zonder ongewenste veranderingen uit te voeren?
- Welke manier om code automatisch aan te passen is het meest geschikt om in deze casus toe te passen?
- Wat zijn statistisch gezien de meest voorkomende problemen bij het updaten van code?

1.3. Onderzoeksdoelstelling

Om de onderzoeksvraag te beantwoorden, wordt als proof of concept een applicatie ontwikkeld die de programmeurs ondersteunt in het updateproces. In de rest van dit onderzoek zal naar deze applicatie verwezen worden als de *updater*.

Buiten de functionele requirements van de updater zal dit onderzoek proberen rekening te houden met de ruimere bedrijfscontext. Dit houdt in dat de gekozen implementatie rekening houdt met de huidige doelgroepen en de middelen/noden van het bedrijf. Concreet betekent dit dat de updater aan de volgende criteria moet voldoen:

- De updater is van de ontwikkelaars voor de ontwikkelaars. De bedoeling is dat de updater gebruikt wordt door de persoon die de update uitvoert.

- De updater moet aanpasbaar zijn aan de uit te voeren update. Het moet herbruikbaar zijn bij toekomstige updates.
- De updater mag geen nieuwe bugs introduceren. Gegeven dat de configuratie correct is, mag het geen ongewenste fouten maken.
- De updater mag niet gebaseerd zijn op Angular. Dit zorgt ervoor dat de updater zelf niet geüpdatet moet worden bij een nieuwe Angular-versie.
- De updater stuurt geen informatie door aan derde partijen. Het bedrijf bevindt zich in de financiële sector, waardoor confidentialiteit een prioriteit is.

1.4. Opzet van deze bachelorproef

De rest van deze bachelorproef is opgebouwd als volgt:
In hoofdstuk 2 wordt een overzicht gegeven van de stand van zaken binnen het onderzoeksdomein, op basis van een literatuurstudie. Hier geven we een omschrijving van wat Angular is en hoe een Angular-project is opgebouwd. Verder overlopen we wat refactoring is en welke manieren reeds bestaan om dit te automatiseren.

In hoofdstuk 3 wordt de methodologie toegelicht en worden de gebruikte onderzoekstechnieken besproken om een antwoord te kunnen formuleren op de onderzoeksvragen. De methodologie begint met het toelichten van de gekozen refactoringtechnieken uit de literatuurstudie. Hierna volgt een korte oplijsting van welke veranderingen concreet uitgevoerd moeten worden om een Angular-applicatie van v16 naar v20 te updaten. Vervolgens wordt als proof of concept de updater uitgewerkt op basis van de gekozen technieken. Tegelijk wordt een gecontroleerde omgeving gemaakt die dient om de effectiviteit van de updater te testen.

In hoofdstuk 4 omschrijven we de uitwerking van de gekozen methodologie. Hier gaan we dieper in op de code achter de updater. Verder worden enkele voorbeelden gegeven van hoe de updater een aanpassing kan automatiseren.

In hoofdstuk 5, tenslotte, geven we de resultaten van de updater. Op basis van deze resultaten geven we een conclusie en formuleren we een antwoord op de onderzoeksvraag. Om het onderzoek af te ronden, worden enkele nieuwe inzichten gegeven. Deze kunnen dienen als aanzet tot verder onderzoek.

2

Stand van zaken

In dit hoofdstuk bespreken we de verschillende technologieën die betrekking hebben op het onderzoek. Deze literatuurstudie start met een omschrijving van het Angular framework en hoe een Angular-project gestructureerd is. Vervolgens overlopen we hoe Angular gebruik maakt van de TypeScript-programmeertaal. We geven een kort overzicht van de nodige veranderingen tussen Angular v16 en v20. Ten slotte volgt een overzicht van verschillende gekende manieren om code automatisch aan te passen.

2.1. Angular

Angular is een user interface (UI) framework ontwikkeld door Google in 2016 (Cincovic e.a., 2019). Het is open-source onder de MIT-licentie en wordt onderhouden door een diverse groep van ontwikkelaars. Angular is de directe opvolger van AngularJS. Hoewel ze dezelfde naam delen, is Angular een volledig nieuw framework met een andere architectuur.

Angular wordt gebruikt voor het ontwikkelen van dynamische webapplicaties. De code van een Angular applicatie volgt een eenduidige structuur. Om deze reden wordt het omschreven als een opinionated framework. Parker (2017) definieert een framework als opinionated als het de ontwikkelaar aanstuurt om op een specifieke manier te werken. Opinionated frameworks houden zich aan strikte conventies die dicteren hoe een project is opgesteld en geschreven. Dit helpt de broncode van een applicatie consistent te houden.

Buiten UI-functies komt Angular met verschillende functionaliteiten die het ontwikkelingsproces ondersteunen (Wilken, 2018). Het komt ingebouwd met een Hyper Text Transfer Protocol (HTTP) client voor een applicatie te verbinden met een backendservice over het internet. Hiernaast heeft Angular een collectie van Command Line Interface (CLI) tools die de ontwikkelaars helpt bij het maken van een

applicatie. Bijvoorbeeld, het genereren van een blanco component met bijhorende testen in één commando. Tenslotte komt het met functies die toelaten om testen te schrijven voor de UI.

Angular is gebaseerd op TypeScript en gebruikt dit in combinatie met andere technologieën:

- De TypeScript-programmeertaal wordt gebruikt voor de implementatie van de bedrijfslogica en testen.
- HTML wordt gebruikt om de structuur van de UI te omschrijven.
- CSS wordt gebruikt om de visuele representatie van de UI te omschrijven
- JSON wordt gebruikt voor het configureren van Angular en TypeScript.

In de volgende secties van dit hoofdstuk bespreken we de werking van Angular in meer detail.

2.1.1. TypeScript

Het Angular framework gebruikt en is geschreven in TypeScript. Als gevolg neemt TypeScript het grootste deel van een Angular applicatie in. Zoals omschreven door Bierman e.a. (2014) is TypeScript een programmeertaal ontwikkeld door Microsoft in 2012. Het is een extensie van JavaScript die een statisch typesysteem toevoegt. Verder heeft het betere ondersteuning voor objectgeoriënteerd programmeren dan JavaScript. TypeScript is een multiparadigma-programmeertaal. Dit betekent dat het verschillende programmeerstijlen ondersteunt, zoals: functioneel, procedureel, objectgeoriënteerd, ...

TypeScript code is niet uitvoerbaar. Om een TypeScript applicatie uit te voeren, moet deze eerst gecompileerd worden. Angular voorziet een compiler om TypeScript naar JavaScript te compileren. Hiervoor maakt Angular achterliggend gebruik van de TypeScript compiler. Volgens Ramos (2024) zijn er twee verschillende manieren waarop een Angular applicatie gecompileerd kan worden. Just-in-Time (JIT) en Ahead-of-Time (AOT). JIT compileert de applicatie in de webbrowser tijdens runtime. AOT daarentegen compileert de applicatie op voorhand en stuurt de output naar de webbrowser. Deze studie focust op applicaties die AOT-gecompileerd zijn. Dit is de standaardmanier van werken sinds Angular v9.

Angular gebruikt TypeScript op een objectgeoriënteerde manier. Het is echter niet strikt objectgeoriënteerd. Angular komt naast klassen met een collectie aan losstaande functies. Een Angular applicatie bestaat voornamelijk uit klassen in combinatie met TypeScript decorators. Martins e.a. (2025) omschrijft decorators als een manier om extra data te koppelen aan een klasse. Decorators kunnen meegegeven worden aan methodes, members, properties of de klassedefinitie. Deze functie is niet uniek aan TypeScript. In Java noemt men dit annotaties en in C# attributes.

Angular maakt gebruik van verschillende decorators voor verschillende doeleinden. Welke decorators en hun werking worden in de volgende secties van dit hoofdstuk in meer detail omschreven.

2.1.2. Componenten

Angular applicaties volgen een componentgebaseerde architectuur (Angular Team, 2025b). Deze architectuur breekt een webpagina op in verschillende bouwblokken, genaamd componenten. Een component kan een klein deel van de UI vormen, zoals een knop of een invoerveld. Of het stelt een groot deel van de UI voor, zoals een navigatiebalk of een formulier. Eén van de belangrijkste eigenschappen van een component is dat deze andere componenten kan bevatten. De compositie van meerdere componenten vormt uiteindelijk de webpagina die de eindgebruiker te zien krijgt. Een component encapsuleert het uiterlijk van een UI-element samen met de achterliggende logica. Voorbeelden van achterliggende logica zijn: valideren van input, data opvragen van een server, ...

Het Angular Team (2025d) omschrijft de opbouw van een component. Componenten zijn TypeScript-klassen met een `@Component` decorator. De `@Component` decorator verwacht drie parameters. Een *selector* die de naam bepaalt van de component. Een *template* dat de structuur van de UI bepaalt. Dit kan ofwel een verwijzing zijn naar een HTML-bestand of een hardgecodeerde string met HTML-code. Tenslotte is er de *style* parameter die het uiterlijk van de UI bepaalt. Dit kan ofwel een verwijzing zijn naar een CSS-bestand of een hardgecodeerde string met CSS-code. Codefragment 2.1 geeft een voorbeeld van hoe deze syntax eruitziet.

```
1 import { Component } from '@angular/core';
2
3 @Component({
4   selector: 'my-component',
5   templateUrl: './my-component.html',
6   styleUrl: './my-component.css',
7 })
8 export class MyComponent { ... }
```

Codefragment 2.1: Voorbeeld van hoe een Angular-component gedeclareerd wordt in TypeScript.

2.1.3. Templates

Angular-componenten maken gebruik van templates om de structuur van de UI te definiëren (Angular Team, 2025h). Templates zijn syntactisch gelijkaardig aan HTML, maar er zijn enkele verschillen. HTML is de standaardtechnologie die webbrowsers gebruiken om de structuur van een webpagina te interpreteren. Volgens

het Angular Team (2025e) breiden template de mogelijkheden van HTML uit met nieuwe syntax specifiek aan Angular. Om HTML dynamischer te maken, voorziet het de `@if`, `@else`, `@for` en `@switch` syntax. De werking is functioneel identiek aan hun TypeScript-equivalent. Codefragment 2.2 geeft een voorbeeld van hoe deze syntax eruitziet. Buiten conditionele syntax is er nieuwe syntax om data uit de component klasse te koppelen aan de template. Dit proces noemt men data binding.

```
<h1>User profile</h1>
@if (isAdmin()) {
  <h2>Admin settings</h2>
  <!-- ... -->
} @else {
  <h2>User settings</h2>
  <ul>
    @for (badge of badges(); track badge.id) {
      <li class="user-badge">{{badge.name}}</li>
    }
  </ul>
}
```

Codefragment 2.2: Voorbeeld van conditionele syntax in Angular-templates.

2.1.4. Data binding

Angular gebruikt data binding als manier om data uit te wisselen tussen TypeScript en templates. Volgens Özdikililer (2021) is data binding een manier om een databron met een bestemming te verbinden. In Angular vormt de TypeScript-klasse de databron, waarbij de template de bestemming is. Conceptueel zijn er twee soorten van data binding: one-way binding en two-way binding. One-way binding zorgt dat een verandering aan de databron gesynchroniseerd wordt met de bestemming. Veranderingen in de bestemming daarentegen worden niet gereflecteerd in de databron. Two-way binding synchroniseert de databron met de bestemming in beide richtingen. Een verandering in de ene wordt gereflecteerd in de andere.

Angular heeft vier verschillende soorten syntax voor data binding. De eerste noemt *interpolation* (Angular Team, 2025c). In de context van templates is interpolation een manier om dynamische tekst te tonen. Het is one-way binding. De tekst kan via de UI niet aangepast worden. De syntax voor interpolation is `{{ }}`, bijvoorbeeld: `<p>Welcome {{ name }}</p>`.

De tweede soort noemt *property binding* (Angular Team, 2025c). Property binding is een manier om dynamisch data aan een HTML-element of een andere compo-

nent door te geven. Het is one-way binding. Het HTML-element of de andere component kan de data enkel lezen. De syntax voor property binding is [ ], bijvoorbeeld:
`<button [disabled]="hasErrors()">Save</button>`.

De derde soort noemt *event binding* (Angular Team, 2025a). Hier is de data een referentie naar een functie. Deze functie wordt meegegeven aan een HTML-element of component. De bestemming van de functie roept de functie op als een bepaalde conditie bereikt is. Deze functie wordt om deze reden een *event listener* genoemd. Het is one-way binding. Het HTML-element of component kan de functie enkel oproepen. De syntax voor event binding is ( ), bijvoorbeeld:
`<button (click)="saveChanges()">Save</button>`.

De laatste soort is two-way binding (Angular Team, 2025j). Two-way binding in Angular is hetzelfde als property binding. Met het verschil dat het HTML-element of de component nu de data kan aanpassen. De syntax voor two-way binding is [( )], bijvoorbeeld:
`<input type="text" [(ngModel)]="firstName" />`.

2.1.5. Services

Het Angular Team (2025f) omschrijft *services* als herbruikbare stukken code die gedeeld kunnen worden over meerdere componenten. Het zorgt ervoor dat verschillende componenten dezelfde logica delen. Dit maakt een applicatie modulair en zorgt ervoor dat code herbruikbaar blijft.

Services worden door de ontwikkelaars van de applicatie gemaakt. Angular verbindt de services met de componenten door middel van dependency injection. Razina en Janzen (2007) omschrijft dependency injection als een ontwerppatroon om van buitenaf logica aan een object mee te geven. Het zorgt voor een losse koppeling tussen objecten, waardoor de code modulair blijft.

```
1 import { Injectable, Component, inject } from '@angular/core';
2
3 @Injectable({ providedIn: 'root' })
4 export class MyService { ... }
5
6 @Component({ ... })
7 export class MyComponent {
8   private service = inject(MyService);
9 }
```

Codefragment 2.3: Voorbeeld van hoe een service in Angular gedeclareerd en geïnjecteerd wordt.

Angular injecteert services in componenten tijdens hun instantiatie. Om Angular

dit voor ons te laten doen, moeten alle services de `@Injectable` decorator hebben. Deze services worden geïnjecteerd in een klasse via de constructor of de `inject()` functie. Hoewel beide opties mogelijk zijn, wordt de `inject()` functie als de standaardmanier beschouwd sinds Angular v18. Codefragment 2.3 geeft een voorbeeld van hoe dit eruit ziet.

2.1.6. Testen

Testen spelen een belangrijke rol in het ontwikkelingsproces. Testen, valideren en verifiëren dat een applicatie voldoet aan de eisen van de gebruiker (Jamil e.a., 2016). Er zijn verschillende vormen van testen, elk met een ander doel. In deze context zijn unit-testen de voornaamste. Angular komt ingebouwd met functionaliteiten om unit-testen te schrijven voor componenten. Olan (2003) omschrijft unit-testen als een manier om een *unit* in isolatie te testen. In objectgeoriënteerde programmeertalen is een unit vaak een klasse. Unit-testen roepen de methodes van deze klasse automatisch één voor één aan en vergelijken de output. In Angular worden componenten als units beschouwd.

Volgens het Angular Team (2025i) komt Angular met CLI-tools om automatisch unit-testen te genereren. Achterliggend wordt het Jasmine-testing-framework gebruikt. Tijdens het opzetten van een Angular-project wordt Jasmine automatisch geconfigureerd. Het is echter mogelijk om een ander testing-framework te gebruiken. Dit onderzoek gaat ervan uit dat het Jasmine-testing-framework gebruikt wordt sinds dit de standaard is.

Angular voorziet in speciale functies om unit-testen te schrijven voor componenten. Deze functies maken het mogelijk om meer dan alleen TypeScript-klassen te testen. Ze bootsen de werking van de applicatie in een webbrowser na. Het laat toe om templates en de data bindings te testen door interacties met de UI te simuleren.

2.1.7. Versies

Volgens het Angular Team (2025g) streeft Angular ernaar een leading-edge en stabiel framework te zijn. Het framework evolueert continu en probeert up-to-date te blijven met de rest van het webecosysteem. Om dit doel te bereiken krijgt Angular om de 6 maanden een nieuwe *major* versie.

Angular-versies worden omschreven in drie delen: *major*, *minor* en *patch* (Angular Team, 2025g). Patches zijn bugfixes aan de achterliggende werking van Angular. Bij deze updates is er geen tussenkomst van de ontwikkelaar nodig. Minor updates voegen nieuwe *kleine* functionaliteiten toe. Hierbij is het de ontwikkelaar die beslist of deze functionaliteiten toegepast worden in zijn of haar applicaties. Tenslotte zijn er de major updates. Major updates voegen nieuwe *grote* functionaliteiten toe. Deze updates verwachten in de meeste gevallen extra input. Dit komt in

de vorm van code refactoren, extra testen uitvoeren, nieuwe API's leren, ...

Dit onderzoek focust op de major updates. Alle verwijzingen naar Angular versies verwijzen naar de major updates.

2.1.8. Aanpassingen tussen v16 en v20

Om een Angular-applicatie te updaten, voorziet het Angular Team (2025k) een handleiding om het proces te ondersteunen. Deze handleiding focust exclusief op aanpassingen die een ontwikkelaar *moet* uitvoeren op zijn applicaties. Het gaat hier vooral over het vervangen van verouderde of verwijderde functies. Enkel de noodzakelijke aanpassingen zijn hier opgelijst. De aanpassingen die we onderzoeken zijn hierdoor gebaseerd op deze handleiding.

Hieruit formuleren we het volgende antwoord op de deelvraag: welke veranderingen moeten uitgevoerd worden om Angular van v16 naar v20 te updaten? Om een applicatie van v16 naar v20 te updaten, zijn volgens deze handleiding 80 verschillende stappen nodig. Hieronder volgt een kleine opsomming van deze stappen:

- Bij elke nieuwe versie verwacht Angular dat de laatste nieuwe versie van Node.js aanwezig is.
- Bij elke nieuwe versie verwacht Angular dat de applicatie de laatste nieuwe versie van TypeScript gebruikt.
- Bij elke nieuwe versie moet een commando uitgevoerd worden om de Angular *core*- en *cli*-afhankelijkheden te updaten.
- Verschillende functies veranderen van naam.
- Verschillende functies worden vervangen door andere functies.
- Verschillende functies worden verwijderd, omdat Angular nieuwe functionaliteiten als standaard toepast.
- Verschillende functies moeten uit andere bestanden geïmporteerd worden.
- Verschillende functies geven nieuwe objecten terug.
- Verschillende functies krijgen nieuwe of andere parameters.
- Door een verandering aan de achterliggende werking van het testing-framework moeten testen die bepaalde functies aanspreken herzien worden.
- In v17 worden conditionele uitdrukkingen in templates strikter geëvalueerd. De werking moet overal nagegaan worden. Dit kan redelijk wat impact hebben op de user interface.
- In v18 is er een aanpassing aan de werking van two-way binding in bepaalde randgevallen.

- In v18 verandert de manier waarop Angular-applicaties van pagina veranderen.
- In v19 en v20 is er een aanpassing aan de werking van interpolation in bepaalde randgevallen.
- ...

2.1.9. Waarom updaten?

Angular-versies zijn tijdsgebonden en komen uit om de 6 maanden (Angular Team, 2025g). Versies zijn niet gebonden aan het aantal of de samenhorigheid van de aanpassingen, zoals bij andere frameworks het geval is. De aanpassingen per versie hebben effect op verschillende aspecten van het framework. Een nieuwe versie kan komen met veranderingen aan bijvoorbeeld: testen, CLI-tools, templatesyntax, animations, ... Het kan ook een aanpassing zijn aan de dependencies van Angular, zoals ondersteuning voor een nieuwere versie van TypeScript.

Niet alle aanpassingen aan Angular zijn zichtbaar in onze code. Dit zijn bijvoorbeeld security- of performantie-updates. Zoals eerder vermeld vallen deze aanpassingen niet onder major updates, maar onder minor updates of patches. Hier is het belangrijk om te vermelden dat sommige functionaliteiten niet meer ondersteund worden bij een nieuwe major update. Deze functies worden verwijderd uit het framework op het moment dat een stabiele vervanging beschikbaar is. Het is belangrijk om op termijn naar de nieuwe functies over te schakelen, omdat deze security-updates blijven krijgen.

Oudere versies van Angular worden niet meer onderhouden. Tijdens het schrijven van dit onderzoek worden v2 tot en met v17 niet verder ondersteund (Angular Team, 2025g). Om de betrouwbaarheid van onze applicatie te kunnen garanderen is het dus van belang om tijdig over te schakelen naar de nieuwste versie.

2.2. Automatisch refactoren

In dit hoofdstuk zoeken we een antwoord op de deelvraag: welke manieren bestaan er om code automatisch aan te passen zonder ongewenste veranderingen uit te voeren? Voordat we hierop kunnen antwoorden, moeten we eerst het concept van *code aanpassen* beter definiëren. In deze context spreken we over refactoren.

Refactoren, zoals omschreven door A. Kaur en Kaur (2016), is het proces om de broncode van een applicatie aan te passen zonder de functie te veranderen. Het doel van refactoren is om de interne structuur van een applicatie te verbeteren. Concreet betekent *verbeteren* voor deze casus dat de applicatie up-to-date is en de conventies blijft volgen van de laatste nieuwe versie van Angular.

De studie door Hodovychenko en Kurinko (2025) vergelijkt verschillende gekende manieren om het refactoringproces te automatiseren. In hun studie worden de manieren onderverdeeld in de volgende categorieën:

- Toolgebaseerd: manuele of semi-automatische technieken ingebouwd in Integrated Development Environments (IDE's).
- Algorithmisch: regel-, patroon- of graaf-gebaseerde algoritmen.
- AI-gebaseerd: machine- of deep learning-modellen.

De volgende secties van het onderzoek geven concrete voorbeelden voor elke categorie. Per voorbeeld worden inzichten gegeven in de werking, samen met de voor- en nadelen. Achteraf wordt de beslissing genomen over welke manier het meest geschikt is om toe te passen in deze casus.

2.2.1. Gekende problemen

Een applicatie refactoren is niet zonder risico's. Bij het aanpassen van software bestaat altijd het risico dat er nieuwe bugs ontstaan. Het is niet altijd even evident om dit op te sporen. Indien een applicatie over voldoende testen beschikt, kunnen bugs snel opgespoord worden. Dan nog is het een meerwaarde om deze bugs op voorhand te vermijden.

De studie door Di Penta e.a. (2020) onderzoekt de kans dat een bepaalde refactoringactie een nieuwe bug introduceert in objectgeoriënteerde applicaties. Uit deze studie blijkt dat acties zoals een methode of variabele van naam of type veranderen de meeste kans hebben om nieuwe bugs te introduceren. Volgens dezelfde bron zijn dit de acties die het meeste voorkomen tijdens refactoring. Extra attentie is nodig bij het automatiseren van deze acties. Dit beantwoordt onze deelvraag: wat zijn statistisch gezien de meest voorkomende problemen bij het updaten van code?

2.2.2. Syntax vs semantiek

Voordat we de verschillende refactoringtechnieken bespreken, is het belangrijk om het verschil tussen syntax en semantiek te begrijpen. Schmidt (1996) omschrijft syntax als het uiterlijk van de programmeertaal. Het bepaalt de structuur en volgorde van verklaringen. Semantiek daarentegen omschrijft de betekenis achter deze structuur. Het bepaalt wat het doel is van een stuk code.

Beide concepten zijn belangrijk in het begrijpen van hoe een applicatie werkt, en of deze *correct* werkt. Als gevolg is het belangrijk om de automatisatietechnieken hierop af te toetsen.

2.2.3. Zoek & vervang

Eén van de simpelste vormen van toolgebaseerd refactoren is de zoek- en vervangfunctie. Zoals de naam het zegt, zoekt dit naar instanties van een bepaalde sequentie van karakters in een tekst om deze vervolgens te vervangen.

De simpelste vorm zoekt op basis van een woord of een vaste sequentie van karakters. Meer complexe implementaties maken gebruik van reguliere expressies (regex). Regex, als omschreven door Goyvaerts (2006), is een speciale sequentie van karakters die een zoekpatroon omschrijft. Het laat toe om naar complexe patronen in een tekst te zoeken.
Zoek- en vervangfuncties hebben meerdere voordelen. Het is goed gekend en gedocumenteerd. Deze functies worden in veel applicaties gebruikt, zoals webbrowsers, tekstverwerkers, IDE's, ... En de meeste programmeertalen komen met functies om dit te implementeren, zowel tekst- als regex-gebaseerd.

Zoek- en vervangfuncties gebruiken op code is niet zonder nadelen. Omdat het enkel zoekt op tekst, heeft het geen vat op de syntax of de semantiek van de programmeertaal. Hierdoor is het mogelijk om ongewenste aanpassingen uit te voeren op de broncode. Bijvoorbeeld: stel dat er twee klassen zijn met de naam *A* en *B*, en dat beide een methode *foo* bevatten. Als we alle instanties van de methode *foo* in klasse *A* willen veranderen naar *bar* met behulp van zoek- en vervangfuncties, zullen alle instanties van *foo* in klasse *B* ook vervangen worden.

Hoewel het theoretisch mogelijk zou zijn om een reguliere expressie te schrijven die met deze specifieke syntax rekening houdt, zijn hier verschillende praktische problemen mee. Uit de studie door Michael e.a. (2019) blijkt dat regexes moeilijk leesbaar, vindbaar, valideerbaar en documenteerbaar zijn.

2.2.4. Language server

Het Language Server Protocol (LSP) is een open protocol ontwikkeld door Microsoft voor Visual Studio Code. Code editors en IDE's gebruiken LSP's om te communiceren met een language server (Bork & Langer, 2023). Een language server is een programma dat programmeertaalspecifieke functionaliteiten aanbiedt. Dit kan in de vorm van het automatisch aanvullen van code, code-diagnostiek, code-navigatie, ... Sinds de ontwikkeling van het LSP is het de facto standaard geworden om deze functies te implementeren in code-editors en IDE's.

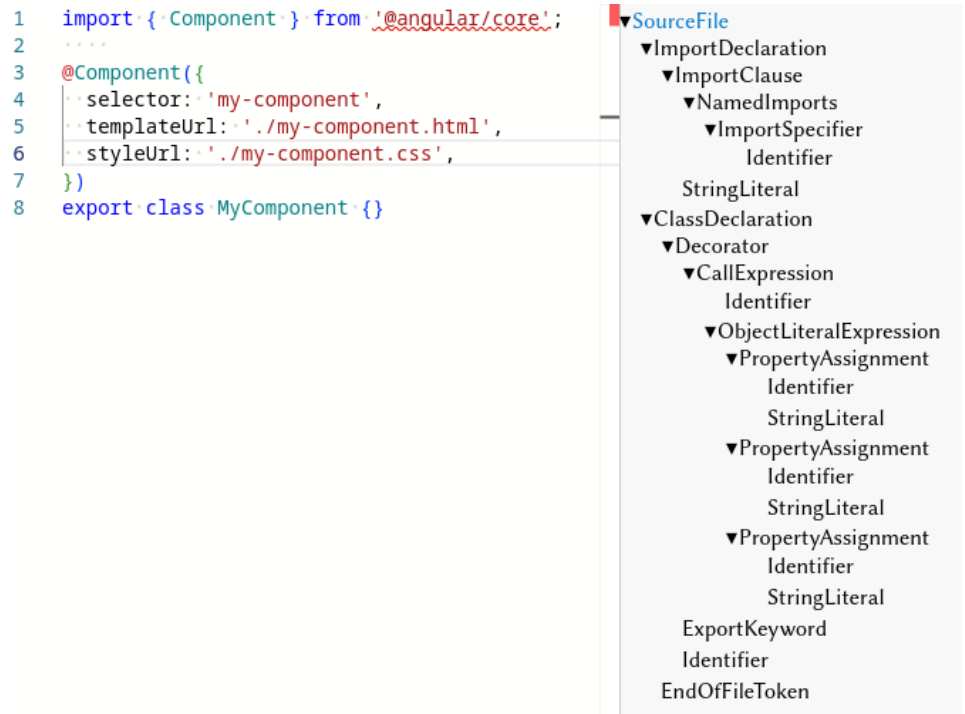
Voor dit onderzoek zijn de TypeScript en Angular language servers relevant. Volgens de TypeScript Language Server medewerkers (2025) biedt de TypeScript language server functies aan die broncode aanpassen. Het kan ongebruikte variabelen verwijderen, imports organiseren, ongebruikte imports verwijderen, ... De Angular language server focust volgens de Angular Language Server medewerkers (2025) op automatisch aanvullen en diagnostiek voor template syntax.

Het voordeel van language servers is dat ze vat hebben op de specifieke syntax van de programmeertaal. In combinatie met een zoek- en vervangfunctie kunnen we gericht code aanpassen. En omdat language servers komen met code-diagnostiek, is het mogelijk om na een verandering direct fouten in de syntax op te sporen.

Language servers zijn echter niet perfect. Ze hebben vat op de syntax, maar niet op de semantiek van de applicatie. Verder zijn we gelimiteerd aan de functies van de language server. Hoewel het mogelijk is om hun functies uit te breiden, is dit in de praktijk niet evident. Het is een zeer niche domein, ondanks dat het de facto standaard is in code-editors en IDE's. Een grondige kennis van het LSP is nodig om een language server aan te spreken of uit te breiden.

2.2.5. Compiler

De studie door Wright e.a. (2013) presenteert een tool voor het refactoren van grote C++ broncodes bij Google. Deze tool maakt gebruik van de Clang C++ compiler om broncode om te zetten naar een Abstract Syntax Tree (AST) om zo de code te doorlopen. Een AST, als omschreven door Sun e.a. (2023), is een datastructuur die de structuur en syntax van een stuk code weergeeft. Dezelfde bron noemt AST's een fundamentele eigenschap van code. Om deze reden wordt deze datastructuur vaak gebruikt om codegerelateerde problemen op te lossen. Figuur 2.1 geeft een vereenvoudigde representatie van hoe een AST van een stuk TypeScript code eruit ziet.



Figuur 2.1: Voorbeeld van een vereenvoudigde AST-representatie (rechts) van een stuk TypeScript-code (links).

Herinner dat Angular gebaseerd is op TypeScript en dat TypeScript gecompileerd moet worden. De TypeScript compiler kan programmatisch aangesproken worden met de TypeScript compiler API. Een studie door Reid e.a. (2023) gebruikt deze API voor het opsporen van foutieve code-elementen, met positieve resultaten.

Net zoals LSP's heeft de compiler vat op de syntax van de programmeertaal. Op het vlak van correctheid is dit één van de meest betrouwbare opties. Verder is de werking van AST's goed gedocumenteerd. Tenslotte bestaan er voor TypeScript verschillende packages die helpen met het opstellen en doorlopen van een AST.

Maar net zoals LSP's heeft de compiler geen vat op de semantiek van de applicatie. Verder is er kennis van boomstructuren en de nuances van de syntax vereist om te kunnen werken met een AST.

2.2.6. Artificiële intelligentie

Met de recente opkomst in populariteit van artificiële intelligentie (AI) zijn reeds verschillende tools ontwikkeld om deze technologie in te zetten voor het schrijven van code. Een studie door Polu (2025) gebruikt AI om automatisch code te refactoren om de performantie van een applicatie te verbeteren. Het toont aan dat deze AI-tools vat hebben op zowel de syntax als de semantiek van een applicatie. Uit dezelfde studie blijkt dat het automatisatieproces correct was in 98% van de gevallen.

AI-gebaseerd refactoren is veelbelovend, maar niet perfect. De studie door Hodovychenko en Kurinko (2025) identificeert enkele praktische problemen met deze aanpak. Om dit soort AI-tools te ontwikkelen, is een grote hoeveelheid betrouwbare data nodig. Het verzamelen en verifiëren van deze data is niet altijd praktisch haalbaar. Refactoringdata voor oude update verzamelen is mogelijk. Maar bij gloednieuwe updates is er nog geen data. Tenslotte is er geen garantie op de correctheid van de output. Een ongewenste aanpassing kan grote gevolgen hebben op de integriteit van de applicatie.

3

Methodologie

In dit hoofdstuk maakt het onderzoek een beslissing over hoe de updater ontwikkeld wordt. Deze beslissing is gebaseerd op de huidige stand van zaken in combinatie met de onderzoeksdoelstellingen. We bespreken de verschillende soorten aanpassingen die moeten gebeuren om een Angular-applicatie van v16 naar v20 te updaten. Vervolgens bespreken we de proof of concept. Hoe de updater geëvalueerd zal worden aan de hand van een testomgeving. Wat de updater kan en doet. En tenslotte bespreken we de validatie van de resultaten. In hoofdstuk 4 wordt de technische uitwerking van de updater in meer detail toegelicht.

3.1. Plan van aanpak

We beginnen met de automatisatietechnieken uit de stand van zaken af te toetsen op de onderzoeksdoelstellingen. Zoek- & vervangfuncties zijn simpel te gebruiken, maar hebben een te grote kans om nieuwe bugs te introduceren. Language-servers kunnen meer context geven en komen met error-detectie. Dit onderzoek observeert een praktisch probleem met deze aanpak. Uit onze observaties blijkt dat er weinig tot geen documentatie is over hoe de TypeScript- of Angular-Language-Servers werken. Dit kan de implementatie moeizaam maken. Compiler-tooling geeft dezelfde functies en is goed gedocumenteerd. Verder toont de studie door Wright e.a. (2013) aan dat deze aanpak werkt op grote schaal. AI heeft de grootste kans op succes, maar er zijn praktische problemen. Verzamelen van data kan problematisch zijn en er is geen garantie op succes.

Op basis hiervan kiezen we ervoor om verder te gaan met zoek- & vervangfuncties in combinatie met compiler-tooling. Voor compiler-tooling gebruiken we de TypeScript Compiler API. Deze API laat toe om op de TypeScript Compiler te programmeren. Het grote voordeel hiervan is dat we toegang krijgen tot dezelfde error-detectie als de compiler. Dit zorgt ervoor dat we nieuwe bugs snel en ac-

curaat kunnen opsporen. Verder is de interne werking, op basis van een AST, goed gedocumenteerd.

De updater moet ingezet kunnen worden voor toekomstige updates. Het probleem hierbij is dat we ons moeten voorbereiden op nieuwe aanpassingen. Dit vereist een flexibele configuratie. Aangezien het doelpubliek programmeurs zijn, stellen we voor om de updater programmatisch op te stellen. Dit geeft ons een hoge flexibiliteit en uitbreidbaarheid. Als programmeertaal kiezen we voor TypeScript. Angular gebruikt dit reeds, dus het is gekend door de programmeurs. Om de integratie met de TypeScript Compiler API te vereenvoudigen, ontwikkelen we een collectie aan helperfuncties. Deze helperfuncties zorgen voor een extra abstractielaag. Het doel hiervan is om snel nieuwe updates te kunnen automatiseren.

De updater zelf zal bij elke aanpassing gehercompileerd moeten worden. Dit klinkt als een complex proces, maar in werkelijkheid betekent dit één extra commando uitvoeren. De updater zal compileren naar een CLI-applicatie. Bij het uitvoeren van een Angular-update moeten er commando's uitgevoerd worden. Door de updater een CLI-applicatie te maken, past deze in de huidige workflow. Verder geeft dit ons de mogelijkheid om alle commando's samen te voegen in een script om de updater op meerdere projecten te laten uitvoeren.

3.2. Soorten aanpassingen

Volgens de Angular update handleiding door het Angular Team (2025k) zijn er in totaal 80 verschillende stappen nodig om een applicatie van v16 naar v20 te updaten. Dit onderzoek verdeelt deze stappen in verschillende *categorieën*. Deze onderverdeling geeft een beter overzicht van wat veranderd moet worden aan een Angular v16 applicatie. Verder geeft deze onderverdeling een genuanceerd inzicht in de resultaten van de updater. Het laat ons toe de sterktes en zwaktes van onze aanpak beter in kaart te brengen. De categorieën zijn opgesteld als volgt:

- Veranderingen aan TypeScript. Dit is het grootste deel van alle aanpassingen.
- Veranderingen aan templates. Dit zijn aanpassingen aan Angular specifieke code in templates.
- Veranderingen aan unittesten. Dit zijn aanpassingen aan de unittests die afhankelijk zijn van Angular.
- Veranderingen aan JSON. Dit zijn aanpassingen aan JSON bestanden die de applicatie configureren.
- Uit te voeren commando's. Dit zijn commando's die uitgevoerd moeten worden in de command line. Meestal gaat dit om Angular packages of dependencies te updaten.

- Veranderingen aan syntax. Dit zijn aanpassingen aan syntax die de werking van de applicatie niet aanpassen.
- Veranderingen aan semantiek. Dit zijn aanpassingen in de achterliggende werking van Angular en/of veranderingen die ervoor zorgen dat de huidige werking van de applicatie moet veranderen.
- Veranderingen die niet van toepassing zijn. Dit zijn aanpassingen aan functies toegevoegd na v16. Het is dus onmogelijk dat de applicaties binnen Stater hiervan gebruikmaken.

Buiten de veranderingen die niet van toepassing zijn, zijn deze categorieën overlappend. In één stap kunnen meerdere categorieën van toepassing zijn. Een verandering kan impact hebben op zowel TypeScript als templates, syntax als semantiek, ...

3.3. Opzet proof of concept

3.3.1. Opzet testomgeving

Dit onderzoek kiest ervoor om de effectiviteit van de updater te testen in een gecontroleerde omgeving om een totaalbeeld te krijgen van alle stappen. Angular is een uitgebreid framework met verschillende functionaliteiten. In de praktijk is het niet zeker of een Angular applicatie al deze functionaliteiten gebruikt. Verder weten we niet op voorhand welke functionaliteiten juist gebruikt worden. Neem bijvoorbeeld animaties. Angular voorziet in functies om animaties toe te voegen aan de UI. Niet alle applicaties hebben dit nodig. Als er updates zijn aan deze functionaliteiten, zal de updater ze niet kunnen uitvoeren, omdat de functie in kwestie niet gebruikt wordt. De updater testen op een willekeurige applicatie kan hierdoor een verkeerd beeld schetsen van wat mogelijk is. Het opzetten van een testomgeving geeft de mogelijkheid om een ruimer beeld te schetsen van wat mogelijk is met de updater.

Om de effectiviteit van de updater te meten, zet dit onderzoek een testomgeving op. De testomgeving is een applicatie gemaakt met Angular v16. Het bestaat uit verschillende klassen en componenten specifiek geschreven met code die moet veranderen in de update naar v20. De applicatie is enkel syntactisch correct, verder heeft het geen doel. Dit wil zeggen dat het enkel moet kunnen compileren zonder fouten, meer niet. Er is bewust gekozen geen verdere semantiek aan de testomgeving te koppelen, omdat de huidige aanpak er geen rekening mee kan houden.

Het volgende proces wordt gehanteerd in het opstellen van de testapplicatie. We doorlopen de Angular update handleiding van het Angular Team (2025k) en evalueren elke stap of deze in aanmerking komt voor automatisatie. Verder is de aard van

de verandering belangrijk. De updater kan syntax interpreteren, maar geen semantiek. Tabel 3.1 geeft een overzicht van welke soort veranderingen in de testomgeving opgenomen worden. Voor elke opgenomen verandering voorzien we een codefragment dat geüpdatet moet worden. Alle codefragmenten zijn semi-realistisch en volgen de Angular best-practices waar mogelijk. Ten slotte geven we per codefragment minstens één stuk code mee dat gelijkaardig is aan de code die veranderd moet worden. Dit dient als controle om na te gaan of de updater specifiek genoeg ingesteld is.

3.3.2. Opzet updater

Dit onderzoek stelt een updater op die de testapplicatie van Angular v16 tot v20 autonoom tracht te updaten waar mogelijk. Buiten het uitvoeren van de updates zal deze updater bijhouden wat het kan en niet kan. Elke stap in het updateproces wordt manueel gecategoriseerd volgens de categorieën besproken in hoofdstuk 3.2. De updater probeert voor elk codefragment in de testapplicatie de nodige aanpassingen te detecteren en vervolgens te automatiseren.

Om een beter inzicht te krijgen in de capaciteiten van de updater worden detectie en automatisatie elk onderverdeeld in drie categorieën: niet, gedeeltelijk en volledig. Tabel 3.2 geeft een overzicht van wat dit concreet inhoudt.

Om de updater te maken, voorziet het onderzoek verschillende helperfuncties om de implementatie te vereenvoudigen. Deze helperfuncties maken gebruik van de ts-morph package. ts-morph is een open-source package ontwikkeld door Sherret (2025). Het is een wrapper bovenop de TypeScript Compiler API die het mogelijk maakt een TypeScript project om te zetten naar een AST. Verder biedt het verschillende functies aan om een AST te navigeren en te manipuleren. Onze helperfuncties vormen een extra laag hierbovenop. Het doel hiervan is om een eenvoudige syntax te creëren, specifiek naar de noden van de updater.

3.3.3. Validatie updater

Om de correctheid van de updater te testen, stelt dit onderzoek een controleomgeving op. De controleomgeving is opgesteld door de testomgeving handmatig te updaten naar v20. We valideren de updater door de output te vergelijken met de controleomgeving. Deze vergelijking gebeurt door alle TypeScript bestanden van beide projecten in te lezen en karakter per karakter te vergelijken. De updater in het proof of concept houdt bij welke bestanden aangepast werden en welk type automatisatie van toepassing is. Tabel 3.3 geeft een overzicht van wanneer de output van de updater correct is.

Soort verandering	In testomgeving	Reden
Bestandstype waar de verandering plaatsvindt		
In HTML/JSON	Nee	De updater heeft geen vat op HTML/JSON.
In TypeScript	Ja	De updater is gemaakt om TypeScript syntax te interpreteren. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
In TypeScript & HTML/JSON	Ja	De updater heeft geen vat op HTML/JSON maar wel op TypeScript. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
Aard van de verandering		
Aan semantiek	Nee	De updater heeft geen vat op semantiek. Dit soort aanpassingen zijn afhankelijk van hoe een applicatie Angular gebruikt.
Aan syntax	Ja	De updater is gemaakt om TypeScript syntax te interpreteren. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
Aan syntax & semantiek	Ja	De updater heeft geen vat op semantiek, maar wel op de syntax. We voorzien een codefragment waarvan we weten dat het geüpdatet moet worden.
Randgevallen		
CLI-operaties	Nee	Voor deze stappen is geen nood aan een speciaal stuk code.
Functionaliteiten toegevoegd na v16	Nee	Het is onmogelijk dat deze functionaliteiten gebruikt worden in de applicaties van Stater.

Tabel 3.1: Omschrijft welke stappen uit het updateproces al dan niet opgenomen worden in de testomgeving. Een stap wordt opgenomen in de testomgeving indien deze voldoet aan het bestandstype en de aard van de verandering.

Detecteer-/automatiseerbaarheid	
Categorie	**Omschrijving**
Niet	Deze aanpassingen zijn niet autonoom uitvoerbaar. Dit kan zijn door de limitaties van de updater of de complexiteit van de aanpassing. Dit onderzoek beschouwt een aanpassing als te complex indien het nodig is om meer dan één AST te doorlopen om de aanpassing te detecteren. Per TypeScript-bestand behoort één AST. Indien er meer dan één AST doorlopen moet worden, betekent dit dat de aanpassing betrekking heeft op meerdere bestanden binnen de applicatie.
Gedeeltelijk	Deze aanpassingen zijn gedeeltelijk autonoom uitvoerbaar. Bijvoorbeeld een functie die een nieuwe naam en extra parameters krijgt. De naam kunnen we automatisch veranderen, maar de extra parameters moeten handmatig ingevuld worden, omdat ze afhankelijk zijn van de context. Het is mogelijk dat uit de automatisatie van deze aanpassingen compiler errors ontstaan.
Volledig	Deze aanpassingen zijn volledig autonoom uitvoerbaar. Hier kan de aanpassing volledig autonoom gedetecteerd en/of uitgevoerd worden zonder compiler errors te veroorzaken.

Tabel 3.2: Omschrijft hoe de capaciteiten van de updater onderverdeeld worden.

Conditie	**Correcte indien**	**Reden**
geen output	Niet automatiseerbaar	Er zijn geen veranderingen uitgevoerd.
output = controle	Volledig automatiseerbaar	Er is een verandering en deze matcht de controle.
output ≠ controle	Gedeeltelijk automatiseerbaar	Er is een verandering, maar deze matcht niet volledig met de controle.

Tabel 3.3: Omschrijft wanneer de output van de updater correct is. De updater in het proof of concept houdt bij welke bestanden aangepast werden en welk type automatisatie van toepassing is.

4

Proof of concept

In dit hoofdstuk geven we een technische omschrijving van hoe de updater werkt. We beginnen met een omschrijving van de helperfuncties. Vervolgens geven we enkele voorbeelden van hoe deze functies samen gebruikt worden om een aanpassing aan broncode te automatiseren.

De eerste functie in codefragment 4.1 helpt om de AST op een uniforme manier te navigeren. Het doorloopt de AST vanaf een gegeven node. Onderliggend gebruikt het een diepte-eerst-in-orde-zoekalgoritme. De parameters van deze functie specificeren twee callbackfuncties. De eerste is een predicaat dat nagaat of een node voldoet aan een bepaalde omschrijving. Indien dit waar is, wordt de tweede callbackfunctie opgeroepen. Deze voert een bepaalde operatie uit op de node. De functie geeft het aantal nodes terug dat aan het predicaat voldoet. Dit maakt het mogelijk om deze functie als een predicaat mee te geven en zo de functie te nesten.

Codefragment 4.2 omschrijft een andere manier om de AST te navigeren. Hier doorlopen we de AST van een gegeven node terug naar de root van de AST. De functie roept zichzelf recursief op tot de gewenste diepte of de root bereikt is.

De rest van de helperfuncties zijn predicaten voor codefragment 4.1. We beginnen met codefragment 4.3. Deze functie gaat na of de tekstrepresentatie van een node een gegeven regexpatroon bevat. De tekstrepresentatie van een node is simpelweg hoe een stuk code eruitziet in de broncode. Bijvoorbeeld, deze functie oproepen op de root node van een AST is hetzelfde als zoeken naar een patroon in het eigenlijke bestand. De functie oproepen op een node die een klassedeclaratie voorstelt is hetzelfde als zoeken naar een patroon in deze klasse. Dit betekent ook dat de tekstrepresentatie van een node de tekst bevat van alle kinderen.

Codefragment 4.4 is een extensie op functie 4.3. Hier gaan we na of de gegeven

```
export function findNodes(
  root: Node,
  predicate: (node: Node) ⇒ boolean | number,
  onMatch: (node: Node) ⇒ void,
): number {
  let matches = 0;
  root.forEachDescendant((node) ⇒ {
    if (predicate(node)) {
      matches += 1;
      onMatch(node);
    }
  });
  return matches;
}
```

Codefragment 4.1: Helperfunctie die de AST vanaf een gegeven node doorloopt. Op basis van de callbackfuncties kunnen gericht aanpassingen uitgevoerd worden op de AST.

```
export function getAncestor(node: Node, count: number): Node |
    undefined {
  const parent = node.getParent();
  if (count ⩽ 1 || !parent) return parent;
  return getAncestor(parent, --count);
}
```

Codefragment 4.2: Helperfunctie die de n-de voorouder van een AST node teruggeeft.

```
export function containsPattern(node: Node, pattern: string):
    boolean {
  const matches = node.getText().match(pattern);
  return matches ≠= null && matches.length > 0;
}
```

Codefragment 4.3: Helperfunctie die nagaat of een patroon terug te vinden is in de tekstrepresentatie van een AST-node.

```
export function deepestInstanceOf(node: Node, pattern: string):
    boolean {
  const matchesCurrent = containsPattern(node, pattern);
  const matchesChild = node.forEachChild((child) ⇒
    containsPattern(child, pattern),
  );
  return matchesCurrent && !matchesChild;
}
```

Codefragment 4.4: Helperfunctie die nagaat of een node de diepste instantie van een patroon bevat in de AST.

node de diepste instantie van een patroon bevat in de AST. Concreet controleert de functie of de huidige node, en geen enkele van de kinderen, het patroon bevat.

Alle functies tot nu toe waren tamelijk abstract en kunnen voor meerdere doeleinden gebruikt worden. Wat volgt zijn enkele functies die specifiek één doel hebben. De eerste van deze functies in codefragment 4.5 gaat na of een node in een bepaalde scope ligt. We doen dit door recursief het syntaxtype van de ouder te vergelijken tot een match is gevonden of de root node bereikt is. ts-morph evalueert het syntaxtype aan de hand van de *SyntaxKind* enumeratie. Voorbeelden van syntaxtype zijn: bestanden, importdeclaraties, klassedeclaraties, expressies, decorators, keywords, ... Als een node aan het syntaxtype matcht, dan wordt deze node teruggegeven.

Codefragment 4.6 definieert een helperfunctie om het type van een node te vergelijken. Dit doen we door de tekstrepresentatie van het type te vergelijken met een gegeven regex patroon. De tekstrepresentatie van het type is simpelweg hoe het type gebruikt wordt in de broncode.

De laatste helperfunctie in codefragment 4.7 dient om na te gaan of een node toegankelijk is vanaf een bepaald type. Deze functie kijkt of de gegeven node aangesproken wordt vanuit een klasse. Indien dit zo is, vergelijken we het type van de klasse via de functie in codefragment 4.6.

Deze functies maken het mogelijk om een AST te doorlopen in enkele lijnen code en gericht code te detecteren. Wat volgt zijn enkele voorbeelden van hoe deze functies samen werken. De aanpassingen die uitgevoerd worden, komen uit de Angular update handleiding door Angular Team (2025k). Het eerste voorbeeld in codefragment 4.8 toont aan hoe we de methode van een bepaalde klasse van naam veranderen. Als predicaat zoeken we de naam van de methode op om vervolgens de naam van de klasse te vergelijken. Indien een node aan het predicaat voldoet, vervangt het de tekst met de nieuwe naam van de methode.

De updater kan meer dan methodes van naam veranderen. Ook alleenstaande

```
export function inScopeOf(node: Node, kind: SyntaxKind): Node |
    undefined {
  const parent = node.getParent();
  if (!parent) return undefined;
  if (parent.getKind() === kind) return parent;
  return inScopeOf(parent, kind);
}
```

Codefragment 4.5: Helperfunctie die nagaat of een AST node in een bepaalde scope zit.

```
export function hasType(node: Node, type: string): boolean {
  const matches = node
    .getType()
    .getText(undefined, TypeFormatFlags.InTypeAlias)
    .match(type);
  return matches !== null && matches.length > 0;
}
```

Codefragment 4.6: Helperfunctie die nagaat of een AST node een bepaald type heeft.

```
export function accessedFrom(node: Node, type: string): boolean {
  const accessProp =
      node.getParentIfKind(SyntaxKind.PropertyAccessExpression);
  if (!accessProp) return false;
  return hasType(accessProp.getExpression(), type);
}
```

Codefragment 4.7: Helperfunctie die nagaat of een AST node opgeroepen wordt vanuit een bepaald type.

```
const project = loadProject();
project.getSourceFiles().forEach((file) ⇒
  findNodes(
    file,
    (node) ⇒
      deepestInstanceOf(node, "mutate") &&
      accessedFrom(node, "WritableSignal"),
    (node) ⇒ node.replaceWithText("update"),
  ),
);
await saveProject(project);
```

Codefragment 4.8: Hernoemt alle instanties van de mutate-methode uit de WritableSignal-klasse met update.

functies zijn mogelijk. Neem codefragment 4.9 als voorbeeld. Hier veranderen we alle instanties van de async-functie uit Angular met waitForAsync. TypeScript gebruikt async als sleutelwoord. Deze instanties mogen niet mee veranderen. Door één lijn code toe te voegen, is het mogelijk om alle instanties van het async-sleutelwoord uit te filteren.

Het is niet altijd mogelijk om de nodige aanpassingen te automatiseren. Dan nog kan het een meerwaarde zijn om deze op te sporen. Bijvoorbeeld, in één van de stappen in het updateproces moeten Angular componenten met de OnPush-verandering detectiestrategie nagekeken worden hoe ze interageren met templates. We weten op voorhand dat templates niet toegankelijk zijn voor de updater. Maar we kunnen wel verandering detectiestrategieën gaan opsporen in TypeScript. In codefragment 4.10 zoeken we alle componenten op met de OnPush-verandering detectiestrategie. Vervolgens tonen we de locatie van deze component binnen het project. Dit tonen we via de bestandsnaam, gevolgd door het lijnnummer in de code.

Dit waren enkele voorbeelden van hoe een updater met behulp van de helperfuncties ontwikkeld kan worden. De predicaten in deze voorbeelden zijn opzettelijk simpel gehouden. De specificiteit van een predicaat is afhankelijk van de complexiteit van het project en de manier waarop de broncode geschreven is. Verschillende bedrijven hanteren intern verschillende manieren om code te schrijven en te structureren. Deze voorbeelden werken perfect in de testomgeving van dit onderzoek. Of deze even goed zullen werken in de praktijk is afhankelijk van het project.

```
const project = loadProject();
project.getSourceFiles().forEach((file) ⇒
  findNodes(
    file,
    (node) ⇒
      deepestInstanceOf(node, "async") &&
      node.getKind() !== SyntaxKind.AsyncKeyword,
    (node) ⇒ node.replaceWithText("waitForAsync"),
  ),
);
await saveProject(project);
```

Codefragment 4.9: Hernoem alle instanties van de async-functie met waitForAsync.

```
const project = loadProject();
project.getSourceFiles().forEach((file) ⇒
  findNodes(
    file,
    (node) ⇒
      deepestInstanceOf(node, "OnPush") &&
      accessedFrom(node, "ChangeDetectionStrategy") &&
      !!inScopeOf(node, SyntaxKind.Decorator),
    () ⇒ console.log(file.getBaseName(), file.getStartLineNumber()),
  ),
);
```

Codefragment 4.10: Zoekt naar elke instantie van de OnPush-methode opgeroepen uit ChangeDetectionStrategy in de scope van een decorator.

5

Conclusie

5.1. Test resultaten

Kruistabel 5.1 en 5.2 tonen de resultaten van de updater uitgevoerd op de testomgeving. Alle rijen buiten *n.v.t.* onder *Verandering* zijn overlappend, zoals besproken in hoofdstuk 3.2. De informatie in deze tabellen wordt per kolom gelezen. Elke kolom bevat zowel een absolute als een relatieve waarde. De relatieve waarde is berekend ten opzichte van de eerste rij in de kolom genaamd *#Stappen*.

Uit het totaal van de 80 uit te voeren stappen blijkt dat 27,5% volledig en 10% gedeeltelijk automatiseerbaar is. Dit is lager dan het verwachte resultaat van 65% uit het onderzoeksvoorstel, zie hoofdstuk A. Eén van de factoren die een rol speelt in de automatiseerbaarheid is de aard van de aanpassing. De updater is gelimiteerd in het opsporen van semantische aanpassingen. In de update van v16 naar v20 waren er meer stappen met impact op semantiek dan op syntax. Uit de stappen met impact op semantiek was amper 2,63% automatiseerbaar. Dit kan verklaren waarom de totale automatiseerbaarheid lager ligt dan verwacht. De totale automatiseerbaarheid had hoger kunnen liggen indien er meer syntactische aanpassingen waren.

Tenslotte willen we de aandacht leggen op de kolom *Testen*. Het blijkt dat 20% van alle mogelijke stappen in deze update impact heeft op testen. 87,5% hiervan zijn aanpassingen aan semantiek. Dit wil zeggen dat er een verandering is in de achterliggende werking. Testen zijn belangrijk om de werking van onze applicaties te waarborgen. Als de update de testen aanpast, is het mogelijk dat deze niet meer betrouwbaar zijn.

Categorie	Totaal		TypeScript		Testen		Syntax		Semantiek	
#Stappen	80	100,00%	50	100,00%	16	100,00%	24	100,00%	38	100,00%
Automatiseerbaar										
Volledig	22	27,50%	10	20,00%	3	18,75%	9	37,50%	1	2,63%
Gedeeltelijk	8	10,00%	8	16,00%	0	0,00%	5	20,83%	5	13,16%
Niet	50	62,50%	32	64,00%	13	81,25%	10	41,67%	32	84,21%
Detecteerbaar										
Volledig	24	30,00%	24	48,00%	6	37,50%	12	50,00%	13	34,21%
Gedeeltelijk	5	6,25%	5	10,00%	0	0,00%	2	8,33%	4	10,53%
Niet	51	63,75%	21	42,00%	10	62,50%	10	41,67%	21	55,26%
Verandering										
TypeScript	50	62,50%	50	100,00%	16	100,00%	17	70,83%	37	97,37%
Template	10	12,50%	3	6,00%	2	12,50%	6	25,00%	4	10,53%
Testen	16	20,00%	16	32,00%	16	100,00%	2	8,33%	14	36,84%
JSON	12	15,00%	0	0,00%	0	0,00%	1	4,17%	0	0,00%
CLI	12	15,00%	0	0,00%	0	0,00%	0	0,00%	0	0,00%
n.v.t.	10	12,50%	0	0,00%	0	0,00%	0	0,00%	0	0,00%
Syntax	24	30,00%	17	34,00%	2	12,50%	24	100,00%	4	10,53%
Semantiek	38	47,50%	37	74,00%	14	87,50%	4	16,67%	38	100,00%

Tabel 5.1: Deel 1 van de resultaten van de updater uitgevoerd op de testomgeving. Alle rijen buiten *n.v.t.* onder *Verandering* zijn overlappend, zoals besproken in hoofdstuk 3.2.

Categorie	Totaal		Templates		JSON		CLI	
#Stappen	80	100,00%	10	100,00%	12	100,00%	12	100,00%
Automatiseerbaar								
Volledig	22	27,50%	0	0,00%	11	91,67%	12	100,00%
Gedeeltelijk	8	10,00%	0	0,00%	0	0,00%	0	0,00%
Niet	50	62,50%	10	100,00%	1	8,33%	0	0,00%
Detecteerbaar								
Volledig	24	30,00%	0	0,00%	0	0,00%	0	0,00%
Gedeeltelijk	5	6,25%	1	10,00%	0	0,00%	0	0,00%
Niet	51	63,75%	9	90,00%	12	100,00%	12	100,00%
Verandering								
TypeScript	50	62,50%	3	30,00%	0	0,00%	0	0,00%
Templates	10	12,50%	10	100,00%	0	0,00%	0	0,00%
Testen	16	20,00%	2	20,00%	0	0,00%	0	0,00%
JSON	12	15,00%	0	0,00%	12	100,00%	11	91,67%
CLI	12	15,00%	0	0,00%	11	91,67%	12	100,00%
n.v.t.	10	12,50%	0	0,00%	0	0,00%	0	0,00%
Syntax	24	30,00%	6	60,00%	1	8,33%	0	0,00%
Semantiek	38	47,50%	4	40,00%	0	0,00%	0	0,00%

Tabel 5.2: Deel 2 van de resultaten van de updater uitgevoerd op de testomgeving. Alle rijen buiten *n.v.t.* onder *Verandering* zijn overlappend, zoals besproken in hoofdstuk 3.2.

5.2. Besluit

De testresultaten tonen aan dat de updater een meerwaarde biedt in de ondersteuning van het updateproces. Ondanks dat het verwachte resultaat niet bereikt is, was het nog steeds mogelijk om een vierde van alle aanpassingen automatisch uit te voeren. De hoge flexibiliteit van onze aanpak maakt het mogelijk om de updater te herconfigureren voor toekomstige updates. Bovendien is de updater niet gelimiteerd aan het uitvoeren van Angular-updates. Dezelfde manier van werken kan toegepast worden om meer algemene refactoringen uit te voeren. Verder kan men de updater configureren om op andere TypeScript-applicaties te werken.

Onze aanpak is gericht op het updaten van meerdere enterprise-applicaties. We zien in dat dit onderzoek niet in alle gevallen een meerwaarde biedt. De reële tijdswinst van deze aanpak is afhankelijk van hoe de updater gebruikt wordt en door wie. De updater configureren om één lijn code aan te passen in één enkele applicatie is contraproductief. Om het meeste uit de updater te halen, moet de tijd voor de updater te configureren kleiner zijn dan de tijd om de update handmatig uit te voeren. Deze rekensom is afhankelijk van verschillende variabelen. Voornamelijk de kennis en ervaring van de persoon die de updater configureert. Iemand met een diepe kennis over de code van het bedrijf zal deze som beter kunnen inschatten dan iemand zonder deze kennis.

In totaal waren er 80 verschillende soorten aanpassingen nodig om een Angular-applicatie van v16 naar v20 te updaten. Zoals omschreven in hoofdstuk 2.1.8 & 3.2 hadden deze aanpassingen betrekking op verschillende aspecten van het framework. Verder hebben we een onderscheid kunnen maken tussen het soort aanpassingen, syntactisch of semantisch.

Er bestaan meerdere manieren om code automatisch aan te passen. Dit kan via tools ingebouwd in IDE's, algoritmes om fouten op te sporen of aan te passen, en machine- of deep learning-modellen. Voor elke manier zijn er verschillende implementaties beschikbaar. In dit onderzoek legden we de nadruk op de meest voorkomende: zoek- & vervangfuncties, language-servers en compiler-tooling.

Zoek- & vervangfuncties in combinatie met compiler-tooling zijn gekozen als de meest geschikte manier om in deze casus toe te passen. Dit onderzoek heeft hiervoor gekozen vanwege de hoge kans op een succesvolle implementatie en de betrouwbaarheid van de output. Zoek- & vervangfuncties zijn welbekend en simpel om mee te werken en te implementeren. Om hun tekortkomingen te compenseren, werd compiler-tooling gebruikt via de TypeScript Compiler API. Meerdere studies tonen aan dat compiler-tooling werkt op grote schaal. Verder geeft het ons dezelfde errordetectie van de compiler, waardoor we op een betrouwbare manier bugs kunnen opsporen.

In hoofdstuk 2.2.1 bespraken we kort wat de gekende problemen waren bij het refactoren van code. In de context van onze aanpak kunnen we zeggen dat aanpas-

singen aan semantiek problematisch zijn om te refactoren. Er is zowel kennis nodig van de werking van Angular als van de applicaties die Angular gebruiken. In de update van v16 naar v20 blijkt dat 47,50% van alle aanpassingen betrekking heeft op de achterliggende semantiek van Angular. Zelfs als een aanpassing betrekking heeft op zowel syntax als semantiek, blijkt dit moeilijk te automatiseren.

5.3. Verder onderzoek

In de stand van zaken hebben we verschillende automatisatietechnieken besproken. Een vergelijking van deze technieken voor toepassing in andere casussen kan waardevol zijn. Wanneer zou een integratie met een language server, of met AI, gepast zijn bijvoorbeeld?

Tijdens het schrijven van dit onderzoek is Angular v21 uitgekomen. Deze versie komt met nieuwe tools om AI beter te integreren in het ontwikkelingsproces. Voornamelijk beweert het Angular Team (2025l) dat het AI toelaat om de nieuwste functionaliteiten te gebruiken. Dit was één van de redenen dat AI niet gekozen werd in dit onderzoek. Deze tools zijn momenteel nog experimenteel, maar kunnen veelbelovend zijn.

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.

Samenvatting

Een applicatie updaten naar een nieuwe versie van een softwareframework kan veel tijd in beslag nemen, zeker als de applicatie meerdere versies achterloopt. Hetzelfde updateproces vervolgens herhalen over meerdere applicaties zorgt ervoor dat de onderhoudstijd snel toeneemt. Het doel van dit onderzoek is om de update van het Angular-webframework van versie 16 naar versie 20 in meerdere applicaties te automatiseren, met als doel de onderhoudstijd voor de ontwikkelaars te verlagen. Framework updates kunnen de performantie en veiligheid van de applicatie verbeteren. Het is daarom belangrijk om deze tijdig uit te voeren. Deze updates zijn niet altijd even simpel om toe te passen, zeker als deze gepaard gaan met veranderingen aan de broncode van de applicatie. In grotere broncodes neemt het proces om alle veranderingen systematisch aan te brengen veel tijd in beslag. Dit onderzoek start met het maken van een oplijsting van alle nodige aanpassingen tussen Angular versie 16 en versie 20. Vervolgens worden de verschillende manieren om aanpassingen automatisch uit te voeren onderzocht. Op basis hiervan wordt een proof of concept ontwikkeld die het updateproces zal automatiseren waar mogelijk. Deze proof of concept wordt getest in een gecontroleerde omgeving. De effectiviteit wordt beoordeeld aan de hand van het aantal nodige veranderingen die het automatisatieproces correct kan opsporen en uitvoeren ten opzichte van het totaal aantal uit te voeren aanpassingen. Dit onderzoek verwacht dat het 65% van alle nodige veranderingen kan automatiseren. Het automatisatieproces zal naar verwachting de nodige tijd om de applicaties te updaten verminderen.

A.1. Inleiding

Het bedrijf Stater is een end-to-end dienstverlener voor zowel hypothecaire als consumentenkredieten. Ze ondersteunen de kredietverstrekker voor de dienstverlening aan consumenten. Binnen het bedrijf zijn er verschillende applicaties die gebruikmaken van het Angular webframework. Angular is een open-source front-end framework, ontwikkeling door Google, gebaseerd op de TypeScript programmeertaal voor de ontwikkeling van dynamische webapplicaties (Cincovic e.a., 2019). Momenteel is Angular versie 20 (v20) de meest recente stabiele versie. Binnen Stater maken de applicaties gebruik van Angular versie 16 (v16). Het bedrijf is van plan de applicaties te updaten naar de recentste versie, Angular v20.
De updates niet uitvoeren is geen optie. Volgens de studie door Vaniea en Rashidi (2016) zijn software-updates nodig, omdat het nieuwe functionaliteiten introduceert, de performantie verbetert en verzekert dat de software compatibel blijft met andere nieuwe software. Verder omschrijft deze bron dat het up-to-date houden van software cruciaal is om de cyberveiligheid te garanderen.
Het grote verschil in versies zorgt ervoor dat het updaten van alle applicaties veel tijd in beslag neemt. Dit is geen eenmalig probleem; volgens Callaghan (2023) krijgt Angular een nieuwe versie om de 6 maanden. De studie door U. Kaur en Singh (2015) beweert dat het onderhouden van een softwareproject gemiddeld 60% van de kostprijs in beslag neemt. Een manier om de tijd voor software-onderhoud in te korten is daarom best interessant. Hieruit komt de vraag: in welke mate kan de automatisering van het updateproces van Angular v16 naar v20, bij meerdere applicaties, de onderhoudstijd voor de ontwikkelaars verlagen? Om deze vraag te beantwoorden zijn de volgende deelvragen geformuleerd:

- Welke veranderingen moeten uitgevoerd worden om Angular van v16 naar v20 te updaten?
- Welke manieren bestaan om code automatisch aan te passen zonder ongewenste veranderingen uit te voeren?
- Welke manier om code automatisch aan te passen is het meest geschikt om in deze casus toe te passen?
- Wat zijn statistisch gezien de meest voorkomende problemen bij het updaten van code?

Gedurende dit onderzoek zal als proof of concept een applicatie ontwikkeld worden die een softwareproject in Angular v16 automatisch updatet naar Angular v20. In de rest van dit document wordt naar deze applicatie verwezen als de “updater”. De updater doorloopt de broncode van een applicatie en maakt een oplijsting van alle nodige aanpassingen en tracht de aanpassing zelf uit te voeren indien mogelijk. Het doelpubliek van de updater zijn de ontwikkelaars die anders deze aanpassingen aan de broncode manueel uitvoeren. De effectiviteit van de updater wordt

bepaald aan het aantal gedetecteerde en opgeloste aanpassingen tegenover het totaal van de nodige aanpassingen.
In de volgende sectie wordt een kort overzicht gegeven van de huidige stand van zaken binnen het probleem- en oplossingsdomein. Hierna volgt een beschrijving van de methodologie, waar de werking en evaluatie van de updater in meer detail beschreven is. Tenslotte worden de verwachte resultaten besproken, waarin een inschatting wordt gegeven van de bevindingen van het onderzoek.

A.2. Literatuurstudie

A.2.1. Uit te voeren veranderingen

De Angular update guide door het Angular Team (2025k) geeft een uitgebreid overzicht van alle aanpassingen die nodig zijn om een Angular-applicatie van v16 naar v20 te updaten. Uit deze bron blijkt dat in totaal 80 verschillende stappen uitgevoerd moeten worden. Deze stappen gaan van het uitvoeren van commando's tot verschillende aanpassingen aan de code.
Zoals omschreven in de studie door Cincović en Punt (2020), is de code in Angular applicaties onderverdeeld in 3 verschillende soorten bestanden:

- TypeScript-bestanden die de bedrijfslogica bevatten.
- HTML-bestanden die de structuur van de user interface (UI) omschrijven.
- CSS-bestanden die de visuele representatie van de UI omschrijven.

De studie door Di Penta e.a. (2020) onderzoekt welke veranderingen aan code de meeste kans hebben om nieuwe bugs te introduceren. Deze studie maakt het mogelijk om een geïnformeerde inschatting te maken van welke aanpassingen geautomatiseerd kunnen worden zonder ongewenste bijwerkingen te introduceren.

A.2.2. Het automatisatie process

Eén van de simpelste manieren om code in bulk aan te passen is het gebruikmaken van zoek- en vervangfuncties gebaseerd op text of reguliere expressies (Regex). De studie van Michael e.a. (2019) omschrijft Regex als een sequentie van karakters die een patroon in een tekst omschrijft. Uit dezelfde studie blijken enkele problemen bij de implementatie van Regex, namelijk dat het moeilijk leesbaar, vindbaar, valideerbaar en documenteerbaar is. Verder kan Regex geen rekening houden met de semantiek van de programmeertaal, aangezien het enkel op tekst gebaseerd is. Om met de semantiek van de programmeertaal rekening te houden, kan gebruikgemaakt worden van een abstract syntax tree. Zoals omschreven door Sun e.a. (2023), een abstract syntax tree is een datastructuur die de broncode van een applicatie illustreert en rekening houdt met de syntax en semantiek van de programmeertaal. In combinatie met zoek- en vervangfuncties laat dit toe om een stuk code aan te passen, enkel als het in een bepaalde context zit.

Herinner dat Angular gebaseerd is op TypeScript. Een bestaande tool voor TypeScript die gebruikmaakt van een abstract syntax tree is de TypeScript Compiler API. De studie door Reid e.a. (2023) onderzoekt hoe de TypeScript Compiler API gebruikt kan worden voor het corrigeren van foutieve codefragmenten. Deze studie raadt aan om de TypeScript Compiler API te gebruiken voor statische code-analyse vanwege de effectiviteit, accuraatheid en mogelijkheid om foutieve code te detecteren.
Een alternatief voor de TypeScript Compiler API dat ook gebruikmaakt van een abstract syntax tree is het Angular Language Server Protocol (LSP). Het LSP, als omschreven door Bork en Langer (2023), is een open protocol voor gebruik in verschillende code-editors of integrated development environments (IDEs) dat programmeertaal-specifieke functies voorziet zoals: automatisch code aanvullen en code-diagnostiek. Dezelfde bron omschrijft LSP's als het de facto standaardprotocol voor de implementatie van deze functies in IDE's.
Tenslotte is het mogelijk om artificiële intelligentie in te zetten om deze aanpassingen te maken. Met de recente opkomst van AI-tools die specifiek gemaakt zijn voor programmeren, is het mogelijk om deze taak uit te besteden aan AI. Er zijn echter problemen met deze aanpak voor deze casus. Uit de studie door Hodovychenko en Kurinko (2025) blijkt dat AI-gedreven tools een gebrek hebben aan transparantie en risico lopen de semantiek van de programmeertaal in de loop van de tijd fout te interpreteren. Verder maakt deze studie de bewering dat voor het maken van dit soort AI-tools er nood is aan een grote hoeveelheid betrouwbare trainingsdata. Het bemachtigen van deze data is problematisch, vooral als het gaat om code die gebruikmaakt van de allernieuwste updates.

A.3. Methodologie

A.3.1. Literatuurstudie

Dit onderzoek start met een uitgebreide literatuurstudie naar de verschillen tussen Angular v16 en Angular v20. De nodige veranderingen worden opgelijst en onderverdeeld in verschillende categorieën. Deze oplijsting bepaalt de capaciteit van de updater. Verder geeft dit een beter overzicht van welke aanpassingen al dan niet geschikt zijn voor automatisatie. Tenslotte zal deze oplijsting dienen als maatstaf om de effectiviteit van de updater op te meten.
Vervolgens wordt onderzocht wat de verschillende manieren zijn om automatisch code te updaten. Eén of meerdere manieren worden verkozen om te implementeren op basis van de volgende criteria:

- Complexiteit van de implementatie. Een voorkeur wordt gegeven aan het gebruiken van bestaande tools boven het ontwikkelen van nieuwe algoritmes.
- Betrouwbaarheid van de output. Zolang de input hetzelfde blijft, mag de output niet veranderen.

- Gebruiksvriendelijkheid. Het moet bruikbaar zijn voor de persoon die normaal manueel de applicaties updatet.

A.3.2. Ontwikkeling van de proof of concept

In deze fase van het onderzoek zal de updater ontwikkeld worden op basis van de voorafgaande literatuurstudie.
De updater heeft als minimum de volgende twee functies: het detecteren van code die aangepast moet worden en de aanpassingen uitvoeren indien mogelijk. Het detecteren van de code speelt een dubbele rol. In eerste instantie is het nodig om de aanpassing op de correcte plaats uit te voeren. Indien de aanpassing niet geautomatiseerd kan worden, zorgt het voor een overzicht van waar alle nodige aanpassingen gemaakt moeten worden.
De updater wordt in een gecontroleerde omgeving getest om de stabiliteit te verzekeren. Deze gecontroleerde omgeving bestaat uit een testapplicatie gemaakt in Angular v16. De testapplicatie bevat een codefragment voor elke vooraf geïdentificeerde stap in het updateproces.

A.3.3. Evaluatie

De effectiviteit van de updater wordt bepaald aan het aantal gedetecteerde en opgeloste aanpassingen tegenover het totaal aantal aanpassingen.
Deze meting wordt uitgevoerd op de gecontroleerde omgeving die gemaakt is in de proof of concept. Dit geeft een totaalresultaat voor alle aanpassingen die theoretisch nodig zijn.

A.4. Verwacht resultaat, conclusie

Op basis van de literatuurstudie en de gehanteerde methodologie verwacht dit onderzoek dat minstens 65% van alle nodige aanpassingen automatisch uitgevoerd kan worden.
Het automatisatieproces zal naar verwachting de nodige tijd voor de applicaties te updaten verminderen. De totale hoeveelheid tijd die in werkelijkheid bespaard wordt, is afhankelijk van verschillende factoren: de ervaring van de programmeur, hun kennis van de broncode, de grootte van de applicaties, ... Deze oplossing zal wellicht meer tijd in beslag nemen als enkel één kleine applicatie geüpdatet moet worden.
Dit onderzoek tracht de beste methode te implementeren voor deze casus op basis van gekende literatuur. Echter, kan het interessant zijn om andere manieren te implementeren en te vergelijken. Verder onderzoek kan uitgevoerd worden naar de performantie, accuraatheid en complexiteit van de verschillende implementaties besproken in de literatuurstudie.

Bibliografie

Angular Language Server medewerkers. (2025). *Angular Language Server* [GitHub repository]. Verkregen oktober 31, 2025, van https://github.com/angular/vscode-ng-language-service

Angular Team. (2025a). *Angular adding event listeners* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/templates/event-listeners

Angular Team. (2025b). *Angular architecture* (Versie 20). Verkregen november 28, 2025, van https://angular.dev/essentials/components

Angular Team. (2025c). *Angular binding dynamic text, properties and attributes* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/templates/binding

Angular Team. (2025d). *Angular components* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/components

Angular Team. (2025e). *Angular control flow* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/templates/control-flow

Angular Team. (2025f). *Angular dependency injection* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/di

Angular Team. (2025g). *Angular releases* (Versie 20). Verkregen november 24, 2025, van https://angular.dev/reference/releases

Angular Team. (2025h). *Angular templates* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/templates

Angular Team. (2025i). *Angular testing* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/testing

Angular Team. (2025j). *Angular two-way binding* (Versie 20). Verkregen november 23, 2025, van https://v20.angular.dev/guide/templates/two-way-binding

Angular Team. (2025k). *Angular update guide*. Verkregen september 8, 2025, van https://angular.dev/update-guide?v=16.0-20.0

Angular Team. (2025l). Announcing Angular v21. Verkregen januari 5, 2026, van https://blog.angular.dev/announcing-angular-v21-57946c34f14b

Bierman, G., Abadi, M., & Torgersen, M. (2014). Understanding typescript. *European Conference on Object-Oriented Programming*, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

Bork, D., & Langer, P. (2023). Language server protocol: An introduction to the protocol, its use, and adoption for web modeling tools. *Enterprise Modelling and*

Information Systems Architectures (EMISAJ), *18*, 9–1. https://doi.org/10.18417/emisa.18.9

Callaghan, M. D. (2023). Upgrading Angular. In *Angular for Business: Awaken the Advocate Within and Become the Angular Expert at Work* (pp. 95–118). Springer. https://doi.org/10.1007/978-1-4842-9609-7_8

Cincovic, J., Delcev, S., & Draskovic, D. (2019). Architecture of web applications based on Angular Framework: A Case Study. *methodology*, *7*(7), 254–259. https://www.eventiotic.com/eventiotic/files/Papers/URL/df6b5054-816e-4bee-b983-663fb87be2cd.pdf

Cincović, J., & Punt, M. (2020). Comparison: Angular vs. React vs. Vue. Which framework is the best choice? *Zdravković, M., Konjović, Z., Trajanović, M.(Eds.) ICIST 2020 Proceedings*, 250–255. https://www.eventiotic.com/eventiotic/files/Papers/URL/50173409-699e-4b17-8edb-9764ecc53160.pdf

Di Penta, M., Bavota, G., & Zampetti, F. (2020). On the relationship between refactoring actions and bugs: a differentiated replication. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 556–567. https://doi.org/10.1145/3368089.3409695

Goyvaerts, J. (2006). Regular Expressions. *Regular Expression*.

Hodovychenko, M. A. H. M. A., & Kurinko, D. D. K. D. D. (2025). Analysis of existing approaches to automated refactoring of object-oriented software systems. *Вісник сучасних інформаційних технологій*, *8*(2), 179–196. https://doi.org/10.15276/hait.08.2025.11

Jamil, M. A., Arif, M., Abubakar, N. S. A., & Ahmad, A. (2016). Software testing techniques: A literature review. *2016 6th international conference on information and communication technology for the Muslim world (ICT4M)*, 177–182. https://doi.org/10.1109/ICT4M.2016.045

Kaur, A., & Kaur, M. (2016). Analysis of code refactoring impact on software quality. *MATEC web of conferences*, *57*, 02012. https://doi.org/10.1051/matecconf/20165702012

Kaur, U., & Singh, G. (2015). A review on software maintenance issues and how to reduce maintenance efforts. *International Journal of Computer Applications*, *118*(1), 6–11. https://doi.org/10.5120/20707-3021

Martins, L., Francisco, P., & Meirelles, P. (2025). An Exploratory Study of Decorators in the TypeScript Programming Language. *Workshop de Visualização, Evolução e Manutenção de Software (VEM)*, 58–68. https://doi.org/10.5753/vem.2025.14563

Michael, L. G., Donohue, J., Davis, J. C., Lee, D., & Servant, F. (2019). Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions.

2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 415–426. https://doi.org/10.1109/ASE.2019.00047

Olan, M. (2003). Unit testing: test early, test often. *Journal of Computing Sciences in Colleges*, *19*(2), 319–328.

Özdikililer, E. (2021). Data Binding in Front End for Web Applications. *The Journal of CIEES*, *1*(2), 31–34. https://doi.org/10.48149/jciees.2021.1.2.6

Parker, H. (2017). Opinionated analysis development. *PeerJ Preprints*, *5*, e3210v1.

Polu, O. R. (2025). AI-Driven Automatic Code Refactoring for Performance Optimization. https://doi.org/10.21275/SR25011114610

Ramos, A. (2024). Advanced Techniques for Angular Performance Enhancement: Strategies for Optimizing Rendering, Reducing Latency, and Improving User Experience in Modern Web Applications.

Razina, E., & Janzen, D. S. (2007). Effects of dependency injection on maintainability. *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA*, 7. https://digitalcommons.calpoly.edu/csse_fac/34/

Reid, B., Treude, C., & Wagner, M. (2023). Using the TypeScript compiler to fix erroneous Node. js snippets. *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 220–230. https://doi.org/10.1109/SCAM59687.2023.00031

Schmidt, D. A. (1996). Programming language semantics. *ACM Computing Surveys (CSUR)*, *28*(1), 265–267. https://doi.org/10.1145/234313.234419

Sherret, D. (2025). *ts-morph* [GitHub repository]. Verkregen november 7, 2025, van https://github.com/dsherret/ts-morph

Sun, W., Fang, C., Miao, Y., You, Y., Yuan, M., Chen, Y., Zhang, Q., Guo, A., Chen, X., Liu, Y., e.a. (2023). Abstract syntax tree for programming language understanding and representation: How far are we? *arXiv preprint arXiv:2312.00413*. https://doi.org/10.48550/arXiv.2312.00413

TypeScript Language Server medewerkers. (2025). *TypeScript Language Server* [GitHub repository]. Verkregen oktober 31, 2025, van https://github.com/typescript-language-server/typescript-language-server

Vaniea, K., & Rashidi, Y. (2016). Tales of software updates: The process of updating software. *Proceedings of the 2016 chi conference on human factors in computing systems*, 3215–3226. https://doi.org/10.1145/2858036.2858303

Wilken, J. (2018). *Angular in action*. Simon; Schuster.

Wright, H. K., Jasper, D., Klimek, M., Carruth, C., & Wan, Z. (2013). Large-scale automated refactoring using ClangMR. *2013 IEEE International Conference on Software Maintenance*, 548–551. https://doi.org/10.1109/ICSM.2013.93