TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ROOPE HAKULINEN

USING IMMUTABLE DATA STRUCTURES TO OPTIMIZE ANGULAR CHANGE DETECTION

Master's thesis

ABSTRACT

ROOPE HAKULINEN: Thesis template of Tampere University of Technology
Tampere University of Technology
Master of Science Thesis, 42 pages
May 2018
Master's Degree Programme in Information Technology
Major: Pervasive computing
Examiner: University Lecturer Terhi Kilamo

Keywords: Angular, TypeScript, JavaScript, change detection, immutable data structures, performance

Angular is the successor of the extremely popular Angular.js framework. While it is based on many of the same concepts, it still is a complete rewrite of Angular.js. Angular has an extremely interesting approach to change detection which differs quite a lot from what other libraries and frameworks have implemented. While extremely performant already by default, it can still be fine-tuned by the developer.

The target of this thesis was to introduce the optimization possibility related to the usage of immutable data structures and the `OnPush` change detection strategy and to demonstrate the performance gain with a reference application. The reference application built is a complete Angular application. It demonstrates a change detection heavy use case where the original version is suffering from a minor performance problem.

It was proven that the usage of immutable data structures with the `OnPush` change detection strategy can have a great positive impact on the performance of an Angular application. The actual gain is dependent on the architecture of the application. In the case of the reference application the gain was 1.21648-fold in frames per seconds compared to unoptimized version.

TIIVISTELMÄ

ROOPE HAKULINEN: Tampereen teknillisen yliopiston opinnäytepohja
Tampereen teknillinen yliopisto
Diplomityö, 42 sivua
Toukokuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Ohjelmistotekniikka
Tarkastaja: Yliopistonlehtori Terhi Kilamo

Avainsanat: Angular, TypeScript, JavaScript, muutoksentunnistus, muuttumattomat datarakenteet, suorituskyky

Angular on erittäin suositun Angular.js-ohjelmistokehyksen seuraaja. Vaikka Angular sisältääkin useita samoja konsepteja, kuin Angular.JS, on se kuitenkin täysi uudelleenkirjoitus alkuperäisestä. Angularin muutoksentunnistus on erittäin mielenkiintoinen, ja se eroaa huomattavasti muiden kirjastojen ja ohjelmistokehysten toteutuksista. Vaikka Angularin muutoksentunnistus on erittäin tehokas jo valmiiksi, on sitä mahdollista edelleen optimoida kehittäjän toimesta.

Tämän diplomityön tarkoituksena on esitellä tällainen optimointimahdollisuus käyttäen muuttumattomia datarakenteita sekä niin sanottua `OnPush`-muutoksentunnistusstrategiaa, sekä esitellä suorituskykyhyödyn mahdollisuus referenssitoteutuksen avulla. Referenssitoteutus on kokonainen Angular-sovellus, joka demonstroi muutoksentunnistuksen kannalta raskasta käyttötapausta, jolla on oletusmuutoksentunnistusstrategialla pieni suorituskykyongelma.

Muuttumattomien datarakenteiden käytön `OnPush`-muutoksentunnistusstrategian kanssa todettiin parantavan muutoksen tunnistuksen suorituskykyä merkittävästi. Varsinainen hyöty on aina kiinni itse sovelluksen arkkitehtuurista. Referenssitoteutuksen kohdalla hyöty oli 1.21648-kertainen näytön päivitysnopeus mitattuna näytettyinä kehyksinä sekunnissa verrattuna optimoimattomaan toteutukseen.

PREFACE

To my mother, Päivi Hakulinen. Without your constant queries about the status of my master's thesis it would have never seen the sunlight. Here it finally is. Thank you.

20.5.2018

Roope Hakulinen

TABLE OF CONTENTS

1. INTRODUCTION

During the last decade there has been a clear shift in the way web applications are implemented. The traditional approach, where the application's state is stored on servers and interaction happens through page (re)loads, has been replaced by the newer approach where state is stored on the client itself and JavaScript is employed to interact with the user and communicate with the servers asynchronously to synchronize the state between client and server.

These applications are often referred to as *Single-Page Applications* (*SPAs*) as they do not perform actual page loads after the initial page is once loaded, but instead use technique called *AJAX* (*Asynchronous JavaScript and XML*) to fetch the data to be shown from web servers and render it on the *user interface* (often just *UI*). Doing this enables them to react more rapidly to the user interaction and offer user experience closer to the native applications people are used to in the mobile and desktop environments.

Single-page applications have required new techniques to manage the state on the client's end and to render (and especially re-render) the data on the *Document Object Model* (*DOM*) representing the web page as seen by the end user. An essential part of this process is the *change detection.* It is the process of identifying the changes on the underlying data model that need to be reflected on the DOM and then applying these changes effectively. The data model is often referred to as application state (or simply *state*). Single-page applications are studied in chapter 2.

Change detection is not a trivial problem to solve and there are multiple approaches taken by different libraries and frameworks, as seen on the chapter 2 when discussing the earlier implementations by various vendors. The methodology has evolved dramatically since the first solutions but so have the browsers. To be more specific, the support for JavaScript especially has improved by a lot. This is thanks to both: new standards and browser vendors. New standards have been drafted by the TC39 committee responsible for the new versions of the ECMAScript standard, on which the JavaScript is based on. The browser vendors, on the other hand, have been adopting the standards usually even already before they have been finalized and published by the committee and have also optimized their implementations to allow more computation to be made on client-side and visually more attractive applications to be created.

Multiple vendors have implemented their own solution to the problem of change detection but the two most widely used are *Angular.js* and *React*. They both provide a unique solution to the problem of detecting changes. Despite the rivalry between the two and the fact

that their solutions differ quite a lot from each other, there is still something to be learned from both of them. The different change detection mechanisms are studied in chapter 2. Many of the modern change detection strategies rely on the concept of immutable data structures. These will be covered also in chapter 2.

Angular is the next-generation version of the extremely popular Angular.js. Even though it builds on top of some of the same ideas as its predecessor it is still a complete rewrite and thus has taken also different approaches on many areas. Angular along with the most important concepts of it related to this thesis will be examined in chapter 3.

Angular has a unique and an efficient approach to the change detection. Still, as change detection is such an essential part of the performance of the application further optimizations on it can improve the user experience a lot. Thus, optimizing Angular's change detection is the topic in which this thesis will dive into by looking for an answer to the research question of whether Angular change detection can be optimized using immutable data structures. Angular's approach to the change detection is introduced in Chapter 3.

Angular applications are always a tree structure of components composed of other components. Using the immutable data structures to optimize change detection can be extremely powerful as it allows complete subtrees of this component tree to be left unchecked under certain conditions. Chapter 4 introduces example Angular application for comparison between non-optimized version and version optimized based on this technique. The gains of optimization are described more in-depth in the Chapter 5. Chapter 6 will tie together conclusions about usage of the immutable data structures to optimize the Angular performance-wise and also the other consequences it has from the perspective of a developer.

2. BACKGROUND

While this thesis concentrates on certain framework and its change detection mechanisms, it is important to further understand the basic ideas behind modern web applications. The term modern web applications here refers to the concept of single-page applications which will be introduced along with the technologies, namely JavaScript, AJAX and JSON, making them possible.

In the following the sections the history of the frontends of the web applications will be introduced after which the technologies for them are introduced. Angular utilizes these underlying technologies and builds on top of many of the concepts made popular by other frameworks and libraries.

2.1 Single-page applications

Traditionally the frontend for a web application has meant a HTML representation sent over to client from the server to be rendered. Then user can interact with this representation by either clicking links or submitting forms. While doing so the user actually tells browser to make a new HTTP request to the server and to render the resulting HTML. Frontend built this way is not obviously very dynamic as changes can only occur once the server sends a completely new HTML page based on user's requests. This is not optimal as first the new HTML representation needs to be generated, then it needs to be sent to browser which then renders it from a scratch again.

The concept of single-page applications is based on modifying the existing representation sent once by the browser. It utilizes JavaScript to add functionality on top of the page. This functionality can include for example reacting to clicks or keyboard input, altering the DOM of the page constructed based on the HTML sent by the server, or making AJAX requests for the server to fetch data dynamically without triggering a full page load (DOM specification) (AJAX). This data fetched can be in any format understood by both ends but the two most used ones nowadays are JSON and XML (RFC 7159) (XML). Out of these the JSON has lately been more widely used in modern web applications and that is also the default one to be used within Angular applications. Thus, the XML will not be covered in this thesis. Next section contains introduction to these technologies mentioned.

The basis for the whole concept of single-page applications is the JavaScript (Mikowski & Powell, 2016). JavaScript is a programming language that is the only language understood by the majority of browsers. While JavaScript can be used as a general-purpose programming language and is also extremely popular on server-side programming, in this thesis the focus is in the features available in the browser environment.

As a programming language, JavaScript can be described as multi-paradigm as it supports for example traits from imperative, functional and object-oriented programming. It is weakly and dynamically typed (Rauschmayer, 2013).

JavaScript was originally developed by Brendan Eich in 1995 who worked for Netscape Communications Corporation at that time (Netscape, 1995). The purpose of the language was to provide dynamic features for NetScape browser's version 2.0. Some parts of the syntax were adopted from the programming language Java (Brendan Eich, 2016). Other than the somewhat corresponding parts of syntax, the Java and JavaScript are not connected to each other and the name JavaScript was chosen only for branding purposes to benefit from the large adoption of Java language.

Nowadays the JavaScript is based on the standard called ECMAScript standardized by Ecma International (Stoyan Stefanov, 2010; Hongki Lee et al, 2012). The work on the ECMAScript standards is open and can be followed in GitHub (ECMAScript proposals). The standards committee, named TC39 (Technical Committee 39), is composed of representatives of major JavaScript runtime vendors and other large companies utilizing the JavaScript (TC39 committee). At the time of writing this thesis there has been 8 releases of ECMAScript and starting from 2015 there is an annual release which will contain all of the features accepted by the TC39 so far (Latest ECMAScript standard). The names of the ECMAScript standards used to follow running numbering starting from 1 but this was also changed with the release of ES6 which was named ES2015 officially followed by the ES2016 the next year. The latest standard release at the time of writing is the ES2017 released in June 2017 (Latest ECMAScript standard).

Because the users are using browsers that do not yet support all the features of later ECMAScript standards the code utilizing such features needs to be transpiled to match earlier standard versions. This process is called transpiling (Rohit Kulkarni et al, 2015). The concept is essentially the same as in compiling high-level language to lower level language. But when both, the source and target language, are high-level languages it is called transpiling instead of compiling. There exists multiple tools to handle this transpilation with the most used one being Babel (Babel).

JavaScript is said to be event-driven language meaning that the execution is based on reacting to events. In browser environment these events can be for example user interaction, HTTP response coming in or timeout triggering. With JavaScript certain code can be tied to be called upon such an event to react somehow. This reaction may involve updating the DOM of the page or making HTTP requests.

HTTP requests can be made without reloading the page. The technique making this possible is called Asynchronous JavaScript and XML (AJAX). While the name implies the usage of XML, it is not actually necessary. The API provided by the browser is still called XMLHttpRequest (XHR standard). With this API new code can be bound to be executed

when the response is received from the server. This block of code can then take further action such as update the user interface.

JavaScript Object Notation (JSON) is a data exchange format (Zia Ul Haq et al., 2015). It is meant to be easily readable and writable for humans while being also machine manageable. It is based on the JavaScript types such String, Number and Object. JSON has become much more popular choice than XML due to its easier format and good support in JavaScript (Tom Strassner). Based on paper Comparison of JSON and XML Data Interchange Formats: A Case Study by Nurzhan Nurseitov et al. (2010) JSON is also superior in performance compared to XML.

The original single-page applications were using just custom-built JavaScript to implement all of the features. This was okay as the applications were small but as they grew bigger, it was inevitable that some libraries were required to manage the applications consisting of multiple different domain objects. Thus, libraries and frameworks started popping up to solve the common problems. The most essential problem these tried to solve was to reflect the state stored in some kind of data structure in the user interface. While it sounds trivial, it for sure is not as will be seen in the next section covering the change detection mechanisms of many popular libraries and frameworks. Optimizing the change detection is also the topic of this thesis.

2.2 Change Detection

Change detection is the process of mapping the application state into the user interface. In case of the web applications this usually means mapping some JavaScript objects, arrays and other primitives into the DOM which is viewable and interactable by the end user. Even though the mapping of the state to DOM is somewhat simple and straightforward, great challenges are faced when trying to reflect the changes that have happened on the state to the DOM. This phase is called re-rendering and it needs usually to be performed each time there is a change on the underlying data model. This is illustrated in Image 1.

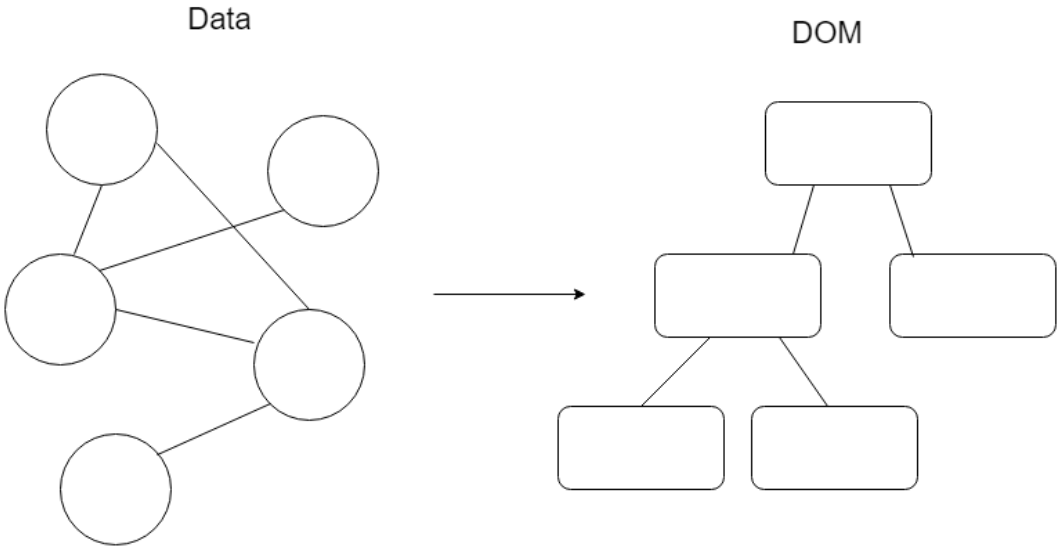

Image 1. Map data to DOM.

The goal of the change detection is to render the data model to the DOM. This also includes re-rendering after changes.

The ultimate goal is to make this process as performant as possible. This goal is greatly affected by the number of expensive DOM accesses needed to fulfill this purpose. But even the minimal possible DOM access does not help if the calculation of changed bindings is slow. These two aspects are the core to understanding the different approaches taken by different libraries and frameworks described next.

The traditional approaches to solve the problem of change detection can be divided into five subcategories based on their evolution level as done by Tero Parviainen (2015) in his commendable article on topic Change and Its Detection in JavaScript Frameworks:

- Server-side rendering
- Manual re-rendering
- Data binding
- Dirty Checking
- Virtual DOM

All of these approaches deserve their own point as they lay down the foundation of change detection to build upon on.

Before the era of SPAs the state was exclusively stored on the backend and all the state transitions happened via navigating through links or submitting forms. Either way, they required a full page load which is obviously slow and does not provide good user experience. This technique is called **server-side rendering**.

No changes can happen in the client. Actions are handled by the browser as user for example clicks a link or submits a form.

Also, nothing is updated in client. New HTML representation is rendered each time on server which is then used to build up the DOM from ground up.

With the raise of JavaScript usage there came also the idea of bringing the data models to the browser instead of just keeping them on server-side. This idea was popularized by frameworks such as Backbone.js, Ext JS and Dojo, which all based on the idea of having events fired on state changes. These events then could be caught by the application developer on the UI code and propagated to the actual DOM. Thus, triggering the actual update is still responsibility of the application developer and can be called **manual re-rendering**.

Change is detected by frameworks defining their own mechanisms to be used for setting the data so that changes can be tracked. For example, the methods for setting the state could be inherited from some framework-dependent class so that once these are called the change detection will be triggered.

When change might have happened, an event is triggered and can be handled by application developer.

First approaches that can be called **data binding** were based on the observation, that the events similar to ones previously introduced, could also be automatically updated to DOM. The main difference to the earlier implementations lays exactly on that there is also support for the receiving end of change events. Example of these approaches is Ember.js.

Even though the UI updating was now "automated", it still had the problem of inconvenient syntax of declaring changes caused by the lack of support by JavaScript. Example of this syntax is presented in Program 1 along with a comparison to the what could be achieved with modern JavaScript such as Proxies introduced in ES6.

```
foo.set('x', 42); // Syntax used
foo.x = 42; // Ideal syntax
```

Program 1.

This kind of awkward, vendor-specific syntax made it possible for solutions to detect the changes automatically with minimal effort. The problem is the obvious binding to the data model of the framework used.

To detect changes, the frameworks have their own functions like set(key, value) which trigger the change detection automatically.

When change might have happened, the changed parts are known as they are always set with a setter function. This makes it possible to only update the changed parts to the UI without comparing what may have changed since last rendering.

Dirty checking is how Angular.js implemented change detection. It was also one of the features that made it so popular among the developers. The basic idea is that for each binding to component's value in the template, a watch function is generated. Every time there might have been a change this set of watch functions built by the template is executed. These watches will then perform check on whether the data has changed since the last time the function was executed and if so, perform an update of changed parts to DOM.

The name dirty checking comes from the process of checking all the bindings every time there is even a possibility of change in the state instead of using approach where framework would be notified about the changes explicitly. This may sound like an expensive operation performance-wise but is actually surprisingly fast as it also minimizes unnecessary DOM accesses that are generally the most expensive part of the update process.

To detect changes custom implementations for possible change sources like setTimeout and AJAX calls ($timeout and $http, respectively, in Angular.js) need to be introduced. If change happens outside of these primitives, the framework needs to be notified manually. In Angular.js for example functions $scope.apply() and $scope.digest() exist to trigger the change detection manually.

When change might have happened all the bound values have the watcher function which keeps the last value in storage and compares it with the current value of binding. If these differ, the new value will be updated to the DOM.

Virtual DOM is an approach made famous by React. In this approach the rendering is done first into a virtual DOM tree, that is still just a vanilla JavaScript data model. This virtual tree presentation can then be converted into corresponding DOM tree. This is what is done initially, but how about when a change occurs?

When a change is detected a new virtual DOM is generated from a scratch. This newly-created structure is then differentiated against the current (virtual) presentation of the DOM. Differentiation patch generated by the comparison is then applied to the actual DOM, thus minimizing the need to touch the actual DOM while still providing extremely easy way to track the changes.

The changes are made in case of React with the this.setState() that needs to be called to set the new state which is then rendered.

When change might have happened a new virtual DOM structure is composed which is then differentiated with the previous structure and the patch generated from differences is applied to the DOM.

As seen above the change detection has evolved radically over the years allowing more complex applications to reside in browsers. The latter two, dirty checking and virtual DOM, are used in the currently most-used frameworks and libraries such as Angular.js (dirty checking), React (virtual DOM), Vue.js (virtual DOM) and also in Angular (dirty checking with some modifications).

2.3 Immutable Data Structures

Immutable data structures are not a new concept. The first references to immutable data structures can be found from the literature describing data structures and functional programming in which context they often are better known (Chris Okasaki, 1998). Here the general concept of immutability is first explored, after which the usage in JavaScript ecosystem is taken a closer look.

2.3.1 The Concept of Immutability

Idea behind immutable data structures is quite simple as the name suggests. That is, once immutable data structure is declared, it cannot be changed. If a new version based on the current data is needed, it can be instantiated by copying the values from original while applying the modifications on top of that. Making a new copy each time data changes sounds like a bad idea performance-wise. It is true that it takes more time than just altering the original, though the performance penalty can be minimized with techniques such as persistent data structure (Driscoll et al., 1986). On the other hand, they have many benefits over mutable data structures.

One of the benefits of the immutable data structures is that they are often found to be simple to rationalize. Second benefit of them is they cannot be altered by mistake. Third benefit is they provide interesting optimization possibilities. Example of such is the possibility to create "a copy" of original object by just pointing to its memory position. This is possible since the data is guaranteed not to change. They are also often used when sharing data between concurrent entities such as threads where mutable data structures might lead to problems with timings. Finally, the main benefit that can be utilized with Angular's change detection is the fact that once data structure changes its memory location must change too. This allows very fast comparison for changes as it is enough to compare the memory addresses of the last and current object.

2.3.2 Immutable Data Structures in JavaScript

ECMAScript 2015 (ES2015 later) has six primitive data types and an `Object` type. The primitive data types are `Boolean`, `Null`, `Undefined`, `Number`, `String` and `Symbol` (Latest ECMAScript standard). Values for the primitive types (such as `true`, 1 and `'my string'`) are immutable whereas each object declared is always a new instance. In the

context of this thesis we are mostly interested in the `Object` type as the applications mostly consists of objects. Arrays are also a special type of an `Object` so the same rules also apply to them.

It is worth noting here that while the ES2015 introduces a new keyword const, it does not make the objects immutable. What `const` keyword does is that it makes sure the variable pointing to object cannot be changed. It does not affect to the object itself but rather the reference to it. Thus, one can write as shown in Program 2.

```
const obj = {};
obj.foo = 'bar';
```

Program 2.

But if a new object were to be assigned to variable marked as `const` and thus the reference would change, `TypeError` would be thrown. This is illustrated in Program 3.

```
const obj = {};
obj = {}; // Uncaught TypeError: Assignment to constant variable
```

Program 3.

ES2015 also introduced `Object.freeze()` method which can be used to freeze an object by not allowing its properties to be changed. This might be useful in some cases but it usually is not enough as the objects pointed by the properties of the original object can still be changed. Hence it is said that it does not deep freeze the object.

Luckily there are multiple solutions made by the community to implement immutable data structures for JavaScript. They differ in APIs they provide and in support for different kinds of data types.

One of the most used ones is the Immutable.js made by Facebook (Adam L. Davis, 2016). Immutable.js is a comprehensive library for immutable data structures for JavaScript, Flow and TypeScript that does support maps and lists among other data types. In this thesis, focus is on Immutable.js' `Map` type. Immutable.js `Map` can be used as described on its home page and shown below in Program 4.

```
var map1 = Immutable.Map({ a: 1, b: 2, c: 3 });
var map2 = map1.set('b', 50);
map1.get('b'); // 2
map2.get('b'); // 50
```

Program 4.

Calls of set return a new instance of `Immutable.Map` with the attribute specified set to the specified value instead of modifying the original `Map`. This way it is ensured that a new instance is always created when a change is applied. But this useful feature comes with the cost of syntax. Using the default JavaScript notation for setting and getting the

object properties does not work anymore, but instead the API provided by the Immutable.js needs to be used to manipulate and query the data.

3. ANGULAR

Angular is a web application framework for single-page applications (Angular). It is partially based on the successful Angular.js framework, though it is a complete re-write. Both - Angular and Angular.js - are open source projects actively developed and maintained by Google.

Angular was initially launched as Angular 2 but when the next major version (version 4 as number 3 was skipped) was about to be released it was advised by the Angular team to drop the version numbering when referring to Angular. This thesis follows the advised guideline to refer to Angular version 1 as Angular.js and to everything starting from Angular version 2 as just Angular.

3.1 Angular.js & Angular

Angular.js was initially released by Google in October 2010 (Angular.js first releases). It quickly gain traction as it provided solutions to many of the problems SPA developers had previously faced with other frameworks and libraries. Angular.js utilizes the popular MVC (Model-View-Controller) pattern to separate presentation, data and logic components.

Angular builds upon on many of the Angular.js concepts but is a complete re-write and has a lot of differences compared to the original Angular.js. Angular's first beta version was published in December 2015 followed by the first release candidate in the beginning of May 2016. During the writing of this thesis the latest release is 6.0.0 which came out on May 2018 (Angular 6 release notes). This version is also used in the reference application.

Angular supports TypeScript, JavaScript and Dart as a programming language. TypeScript is recommended by the core team and is most widely used (TypeScript recommendation). It will also be used throughout this thesis. Though, most of the code provided here is also valid JavaScript as many of the TypeScript features are not meaningful in this context and as TypeScript is a superset of JavaScript.

3.2 Core Concepts

Angular is a complex system with various concepts. The dive-in here is not going to cover many concepts such as directives, pipes or reactive programming as they fall out of the scope. Instead what is covered here, is the overall architecture of an Angular application and the concept of components along with the data binding in templates. Also the very basics of services are explained to understand the reference application better.

Every Angular application must contain a root module class annotated with @NgModule annotation. The root module contains a tree of components with a single root component serving as the start point for bootstrapping. This tree structure is illustrated in the Image 2 below.

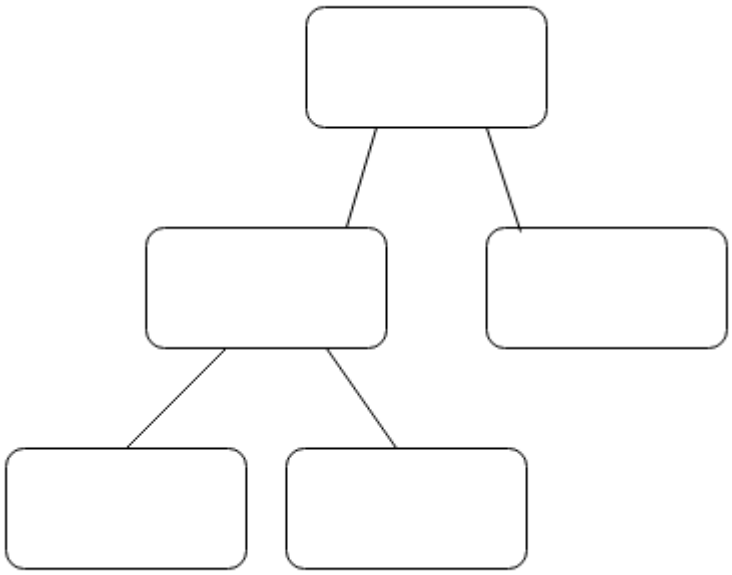

Image 2. Angular component tree structure.

3.2.1 Components

Components are the basic building blocks of the application UI. `@Component` annotation accepts an object containing metadata about the component. Out of this metadata the most important ones are template (or `templateUrl`) and selector. Metadata field template contains the template (usually HTML) that is used to construct the DOM when the component is rendered. If `templateUrl` is used instead it should point to an external file containing the template.

Metadata field selector contains a unique identifier for this component to be used in the templates of other components as a marker where to place this component's constructed representation. This way the selector enables the building of a component hierarchy. Program 5 is an example of such a component.

```
@Component({
  selector: 'app-todos',
  template: `<div>Todos component</div>`
})
export class TodosComponent {
}
```

Program 5.

3.2.2 Templates

Templates are essentially just plain HTML with additional syntax added on top of them that Angular knows how to handle. This additional syntax is used to show data, apply structural changes, change attributes and to bind to DOM events. These additions contain the following:

- Data bindings: The simplest part of the template syntax. They simply show the latest values to the UI. They re-render the value to UI every time it changes on the component. See Program 6 for an example data binding.

```
{{field}}
```

Program 6.

- Structural directives: Named after their behavior to alter the structure of the template. The two most usually seen structural directives are `*ngFor` (renders each item in a list) and `*ngIf` (renders content conditionally). More structural directives can also be implemented by developer. Example of usage of structural directives can be found in Program 7.

```
<div *ngFor="let item of items; let i = index"></div>
```

Program 7.

- Attribute bindings allow setting values of DOM attributes. The idea is the same as for the data binding but instead of showing the data in UI, the attribute value is set. Program 8 shows an example of these bindings.

```
<input [disabled]="property" />
```

Program 8.

Data can also be passed to child components using the same notation. The acceptable attributes for the child component need to be declared as fields of the component class with `@Input` annotation as shown in Program 9.

```
@Component({
  selector: 'app-todo',
  template: `{{text}}`
})
export class TodoComponent {
  @Input()
  text: string;
}
```

Program 9.

which could now be populated in parent component's template as Program 10 illustrates.

```
<app-todo [text]="'Some task'"></app-todo>
```

Program 10.

- Event bindings allow binding of DOM events of a certain element. The events are the basic DOM events from the DOM specification such as change for input elements or click for any element (DOM specification). Event bindings are shown in Program 11.

```
<div (click)="clickHandler()"></div>
```

Program 11.

Whereas the inputs can be used to pass data to the children components, outputs can be used to send events back to the parent component. Outputs are declared with `@Output` annotation of component's fields and are of type `EventEmitter`. `EventEmitter` is a type declared by Angular that extends RxJS `Observable` by adding it a method called `emit` to trigger an event for the parent. The type parameter passed to the `EventEmitter` tells which type the events will be of. Program 12 shows a demonstration of how the outputs can be declared.

```
@Component({
  selector: 'app-todo',
  template: `<button (click)="clicked.emit('A')">Button A</button>`
})
export class TodoComponent {
  @Output()
  clicked = new EventEmitter<string>();
}
```

Program 12.

This could now be caught on the parent template as in Program 13.

```
<app-todo (clicked)="todoClicked($event)"></app-todo>
```

Program 13.

- Template-local variables: Template-local variables can be used to bind variable name for an element. Once declared, the variable can be referenced from within the template. Templates are always bound to a certain component and when they reference some name not declared in template itself (template-local variable), it references to the related component's field or method of same name. Program 14 shows how these can be used.

```
<input #myInput />
{{myInput.value}}
```

Program 14.

3.2.3 Component Lifecycle Hooks

Components are created by Angular as they appear in the templates (besides the root component, obviously) of other components. What this essentially means is that the lifecycle of a component is managed by the framework. Angular provides hooks for all of the events in the lifecycle of a component for developer to implement. The available hooks are (Angular lifecycle hooks):

- `ngOnChanges`: Called when Angular (re)sets input properties for the component.
- `ngOnInit`: Called once Angular has initialized the input properties.
- `ngDoCheck`: Called on each change detection run.
- `ngAfterContentInit`: Called once the content children have been initialized for this component.
- `ngAfterContentChecked`: Called everytime Angular checks the content projected. Called once after the `ngAfterContentInit` and on every subsequent `ngDoCheck`.
- `ngAfterViewInit`: Called once the view children have been initialized for this component.
- `ngAfterViewChecked`: Called every time Angular checks the view. Called after the `ngAfterViewInit` and every subsequent `ngAfterContentChecked`.
- `ngOnDestroy`: Called when the component is about to be removed.

The order of invocation for the lifecycle hooks is described in Image 3.

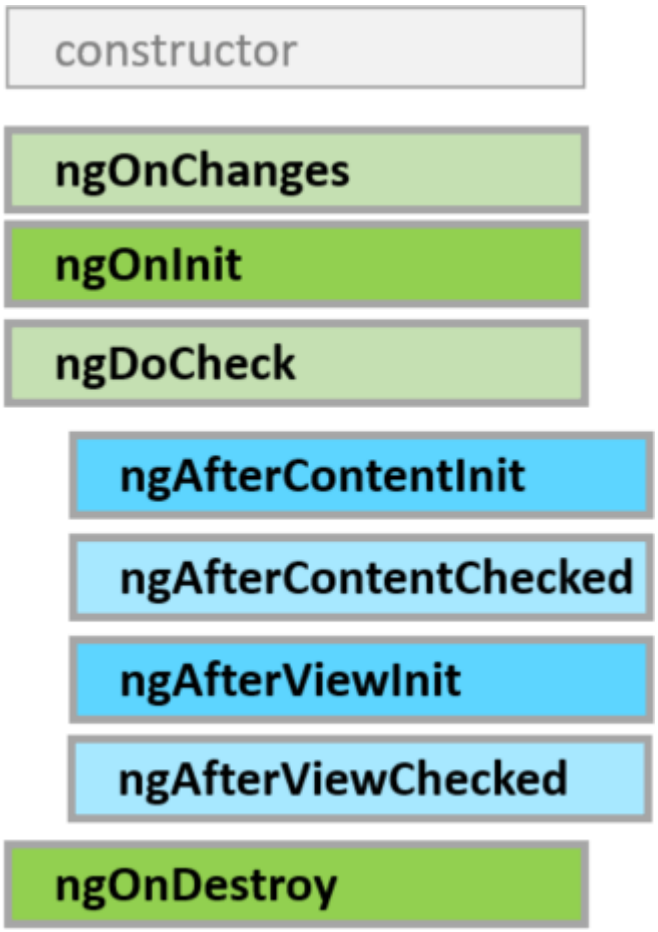

Image 3. Angular lifecycle hooks invocation order (Angular lifecycle hooks)

Each lifecycle hook also has a corresponding interface available. This interface is advisable to be implemented as it allows static type checking of the lifecycle methods of the component.

3.2.4 Services

Beside components one important concept an Angular module can include is a service. Services are singletons within their scope. The scope is based on where they are included but usually they are included in a module and thus the module forms the scope. This is also the case for the services in the reference application of this thesis. Like for components, Angular manages the lifecycle of the services.

Services are useful for many different things. First, they can act as a storage for the duration of application's lifetime as they work as singletons within the app. Second, they can be used to share data between components that cannot communicate via the child-parent relationship. Third, they can be used to contain some common functionality such as the communication with the server.

Services are implemented as classes. These service classes are annotated as `@Injectable`. While this is not mandatory for non-transitive services (services utilizing other services), it still is a good practice to mark them always to prepare for future use cases. The service instance can be accessed by the components and other services of the scope (module in this case) via the dependency injection provided by the Angular.

3.2.5 Dependency Injection

Dependency injection is used to allow easy access to other pieces of the application. Especially it is useful for having singleton services. To register a service as injectable it needs to be listed in the providers field (array) of `@NgModule` annotation. This way Angular knows it exists and can provide it when asked for.

Now all of the different types of concepts - such as services and components - of an Angular application can get access to the injectable items by marking them as their constructor parameters as in Program 15 where `MyService` is the name of the class declared in the `@NgModule` annotation's providers.

```
@Component({
  selector: 'app-todo',
  template: `<div></div>`
})
export class TodoComponent {
  constructor(private myService: MyService) {
      this.myService.doSomething();
  }
}
```

Program 15.

3.3 Angular's Change Detection

Angular takes a unique approach to change detection on many aspects. Just like with other frameworks the change detection mechanism is built by solving the two essential points: how possible change is noticed and what happens when changes might have occurred.

Let us first take a look at the approach used to notice the changes. To understand the mechanism, JavaScript's asynchronous nature needs to be addressed first. The essential part is the fact that JavaScript is, by nature, asynchronous, yet single-threaded.

3.3.1 Execution Model of JavaScript

JavaScript program's execution works normally just like in any other imperative language. Instructions are executed and on function calls the call stack is populated. Once there are no instructions left to be executed, the execution will halt.

Along with this traditional call stack, JavaScript also has something called callback queue. Callback queue can contain functions (and thus more code) to be executed when the traditional call stack gets empty. The only way callback queue can be populated is via set of platform-specific APIs. Platform-specific APIs depend based on which platform, such as Node.js or browser, the code is being executed. They provide a set of APIs to interface with the outside world. For Node.js applications this means for example accessing file system and on browser DOM can be used via the APIs. There are also many APIs that work the same way on most platforms such as timer APIs. Since the focus of this thesis is on the user interface instead of server-side programming only web application APIs are considered here.

For web applications these APIs are called Web APIs (Web APIs). For example, one often used Web API is `setTimeout(fn, timeout)` which takes two parameters: `fn` which is the callback function to be pushed to the callback queue once the delay specified by the second parameter, `timeout`, has elapsed. `setTimeout` is only one of the Web APIs available with more being published all the time by the W3C. Few other examples include making HTTP requests (AJAX), manipulating the DOM and using web socket connections.

In web browsers, these Web APIs can be found from a special object named window which is always available as the global context when code is ran on the browser. window is also the default context from which variables are looked for if no other object is specified. Thus the invocations `setTimeout` and `window.setTimeout` are equivalent in that sense that they both look for method `setTimeout` from window object.

Each platform-specific API call does something outside of the JavaScript execution context. If the operation performed by the API is asynchronous, a function can be passed to be executed once the operation is completed. These functions passed to asynchronous APIs are called micro tasks and they are executed every time the call stack gets empty. Since there can be multiple micro tasks scheduled while previous micro task is being executed there needs to be a queue for storing the micro tasks.

3.3.2 Zone.js

As the Web APIs are just plain functions on the window object, they can be overridden by declaring another function with the same name like: `window.setTimeout = function(fn, timeout) { ... }`. Zone.js is a library published by Angular team that utilizes this possibility to replace all (supported) Web APIs with its own implementation which then acts as a proxy to the actual implementation by browser (Zone.js). While adding this additional proxy layer Zone.js also allows tracking of execution of these methods and micro tasks generated by them. This additional layer is called a zone.

Zones are a concept made famous by the Dart programming language (Dart). As declared on Zone.js README description: "A Zone is an execution context that persists across async tasks." (Zone.js). Besides allowing to track the context among asynchronous tasks, the Zone.js also allows hooking into certain events related to them such as the start of execution of micro task.

Angular implements a new class called `NgZone` on top of Zone.js that extends the API with one essential hook called `onMicrotaskEmpty`. `onMicrotaskEmpty` is called when there are no more items (micro tasks) on the callback queue. This hook is essential for the change detection as seen soon.

3.3.3 Noticing the Changes

As it turns out, the only sources for a change, after Angular component is initialized, are the possible micro tasks that get executed as response to a something happening outside of the execution context of the script. This leads to the great epiphany that the change detection is only necessary when `onMicrotaskEmpty` is triggered since at that time there are no more micro tasks to be executed and thus some changes might have happened to the data and no more coming straightaway.

This approach is a great improvement over the Angular.js where it was necessary to use custom APIs like `$timeout` and `$http` for the framework to be able to keep track of changes for asynchronous code. Another option was to explicitly notify about changes by calling the `$scope.$digest`. The way Angular implements the change detection makes the whole process transparent from the developer's perspective which leads to looser coupling with the framework used as the native Web APIs can be used without additional layer.

3.3.4 Reacting to Possible Changes

Once it is known that the changes might have happened, all the values bound on templates need to be checked for changes. To do this, each component in Angular application has its own change detector. These change detectors form a tree structure just like the actual components do as illustrated in image 4 below.

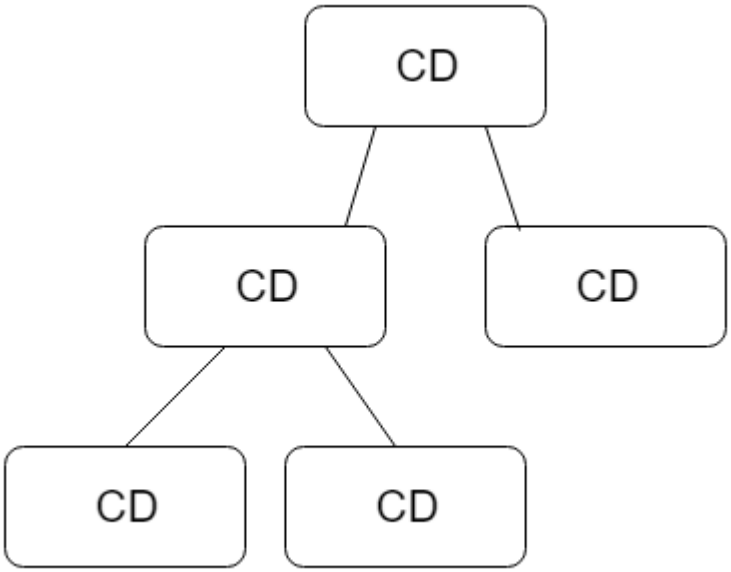

Image 4. Angular change detector (CD) tree.

Each change detector is responsible for updating the bindings found in the template of its associated component. To detect the changes as effectively as possible Angular compiles change detectors based on the actual template instead of using generalized solution. This is a major difference when compared to Angular.js.

For the compilation of templates there are two different options: in the browser when application is loaded or via command line utility as a static build step. These options are called Just-in-Time (JiT) and Ahead-of-Time (AoT), respectively. The first one is used during the development since its fastness, but it lacks essential features required by production builds such as type checking and optimizations. The latter is used for production builds and includes the type checking and optimizations lacking from JiT mode.

The tree structure of change detectors allows Angular to run change detection always top-to-bottom. This is possible because the values of parent can only be used by children via the well-defined inputs (`@Input` annotation). Because of this there is no need to run multi-pass change detection as every piece of data used by component will be either already checked for changes by the parent or is owned by the current component and thus will be responsibility of the component on hand.

Change detectors are just JavaScript code generated based on the templates of components that can be used to check if any value within the component's data model has changed since the last run. If there has been a change the relevant parts are updated in the user interface.

The change detection code could be generic in a meaning that it would use the same code to check for changes in any kind of data. This is actually how Angular.js implemented the change detection. Nowadays, better results can be achieved by usage of generated template-specific code that is optimized for the types of bindings of a certain template.

The code generated for change detectors in Angular is said to be monomorphic (Xavier Leroy, 1997). What this means is out of scope of this thesis, but it makes the detection of value changes extremely efficient by utilizing specific knowledge on how JavaScript interpreters work. Without the monomorphic code it would not be possible to check bindings in so little time as Angular can.

As discussed above the compilation from templates to change detectors is done by default on run-time when the application is loaded by the browser. Because this makes the startup time longer the templates can also be compiled "offline" with command line utility and thus the compiler itself does not even need to be shipped to the browser. This command line utility is called *ngc* (found in npm module *@angular/compiler-cli*) and it is just a thin wrapper on top of *tsc* (standard TypeScript compiler) that can traverse through the templates and generate the necessary change detectors beforehand thus significantly reducing the time needed for the startup of application in browser.

It is also worth noting here that a programmer using Angular can cause a situation where the data of component is changed during the change detection. In this situation the change would not obviously be reflected in the user interface as it is not noticed anymore by change detectors. To catch this kind of incorrect behavior Angular runs change detection twice on each iteration, unless it is set to run in production mode, with the latter round throwing an error on changed values. This is for developer's comfort and in production this would be unnecessary overhead and thus is not done. An example of how this kind of problem can be met is by utilizing the `ngAfterViewInit` lifecycle hook to update a field passed for children as an input as at that point the value has already been set for the child component but the new update does not trigger the change detection to be run again.

In case there is a need for more explicit control of the change detector a component can ask the dependency injection of Angular to provide it with `ChangeDetectorRef` which is a reference to the change detector generated for the component on hand. `ChangeDetectorRef` and its control methods will be investigated further later.

To understand the Angular's change detection better let us examine the steps taken on each iteration of change detection based on Maxim Koretskyi's (2017) article *Everything you need to know about change detection in Angular*. As discussed already earlier these steps are first performed for the root component of the application which again calls the change detectors of the child components thus constituting the tree structure of change detectors.

In Angular term *View* is used to describe a basic building block of a user interface. The definition from Angular's source code (Angular ViewRef code) is as follows:

"A View is a fundamental building block of the application UI. It is the smallest grouping of Elements which are created and destroyed together."

"Properties of elements in a View can change, but the structure (number and order) of elements in a View cannot. Changing the structure of Elements can only be done by inserting, moving or removing nested Views via a ViewContainerRef. Each View can contain many View Containers."

The `ChangeDetectorRef` mentioned above is actually an abstract class which `ViewContainerRef` implements. When the dependency injection is asked for token `ChangeDetectorRef` an instance of `ViewContainerRef` is actually returned.

Views store the current state of change detection as flags. These flags are:

- `FirstCheck`: Whether the next check is the first one.
- `ChecksEnabled`: Whether checks are enable for this view.
- `Destroyed`: Whether the view is destroyed.

These flags are used to determine whether the change detection should be run or not on next iteration, or if there has been an error within the last run. These flags and the behavior related to them can be implicitly controlled by setting a change detection strategy with the `changeDetection` attribute of `@Component` annotation.

For example, change detection is skipped in case `ChecksEnabled` is false or view has `Destroyed` as true. By default, every view is instantiated with `ChecksEnabled` being true.

The 11 steps taken on each component's change detector on each iteration define the Algorithm 1 (Maxim Koretskyi, 2017).

1. update input properties on a child component instance
2. update child view change detection state (part of change detection strategy implementation)
3. call `OnChanges` lifecycle hook on a child component if bindings changed
4. call `OnInit` and `ngDoCheck` on a child component (`OnInit` is called only during first check)
5. call `AfterContentInit` and `AfterContentChecked` lifecycle hooks on child component instance (`AfterContentInit` is called only during first check)
6. call `OnDestroy` if the child/parent component is destroyed
7. update DOM for the current view if properties on current view component instance changed
8. run change detection for a child view (repeats the steps in this list)

9. call `AfterViewInit` and `AfterViewChecked` lifecycle hooks on child component instance (`AfterViewInit` is called only during first check)
10. disable checks for the current view (part of change detection strategy implementation)
11. set `FirstCheck` to false

Algorithm 1.

So essentially the change detector updates its children components' inputs, calls the necessary lifecycle hooks and updates the DOM while maintaining its own state. The different change detection strategies available in Angular are introduced later.

The change detector can be controlled via the `ChangeDetectionRef` of the component. There are five methods to control the change detection:

- `detach`: Sets the `ChecksEnabled` of change detector to false thus preventing further checks.
- `rettach`: Does the reverse of detach method so sets the `ChecksEnabled` to true thus enabling the further checks.
- `detectChanges`: Runs a single iteration of Change detection.
- `markForCheck`: Traverses through the tree of Views and sets `ChecksEnabled` to true for each.
- `checkNoChanges`: Runs the steps 1, 7 and 8 from above and throws an exception if it finds something has changed.

These are powerful operators that can be used to specify more advanced flows for change detection. For example one could only trigger the change detection only if an input has been set to certain value. This is illustrated below in Program 16.

```
@Component()
export class MyComponent implements OnChanges {
    @Input()
    doCheck = false;

    constructor(private cdr: ChangeDetectorRef) {
        this.cdr.detach();
    }

    ngOnChanges() {
        if (this.doCheck) {
            this.cdr.markForCheck();
        }
    }
}
```

Program 16.

The change detection is turned off on the constructor. Since per the steps introduced earlier, the `ngOnChanges` is called before actually detecting the changes, it can be used to enable the change detection for this run of change detection.

To conclude the benefits of the approach of Angular it can be said that it is predictable, easy to debug & rationalize and performant. Guaranteed single-pass, top-to-bottom change detection cycle prevents the common performance bottlenecks known from Angular.js (Miguel Ramos et al., 2017) (Thomas Nagele, 2015).

3.4 OnPush Change Detection Strategy

As seen earlier, Angular's change detection is extremely well controllable. It can be manipulated in a few different ways. First, it can be turned manually on and off as developer sees beneficial via the `ChangeDetectorRef` available for each component. Second, a change detection strategy can be set.

The change detection strategy describes how the changes to component's data structures should be detected. It can be specified as part of `ComponentMetadata` passed to `@Component` annotation. The possible options are `Default` and `OnPush`. `Default` is the normal mode while `OnPush` only performs full change detection once any input of component has been set to point to a new value or an event (like `click`) has happened within the component. Program 17 shows how to enable `OnPush` strategy for a component.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class AppComponent {

}
```

Program 17.

Since `OnPush` only performs full change detection once any input of a component has been set to point to a new value it can be leveraged with the immutable objects that work exactly this way by providing always a new copy of the data. This way the whole process of change detection can be skipped for the whole subtree of the component if the inputs are the same.

To provide a better understanding of how the `OnPush` strategy works let us examine the Programs 18, 19, 20 and 21.

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
```

```
export class AppComponent {
  people = [
    {
      name: 'John'
    },
    {
      name: 'Jane'
    }
  ];

  updatePeople() {
    this.people.splice(0, 2, ...[
      {
        name: 'David'
      },
      {
        name: 'Taylor'
      }
    ]);
  }

  replacePeople() {
    this.people = [
      {
        name: 'David'
      },
      {
        name: 'Taylor'
      }
    ];
  }
}
```

Program 18. *app.component.ts*

```
<app-people [people]="people"></app-people>
<button (click)="replacePeople()">Replace</button>
<button (click)="updatePeople()">Update</button>
```

Program 19. *app.component.html*

```
@Component({
  selector: 'app-people',
  templateUrl: './people.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class PeopleComponent {
    @Input()
    people: any[];
}
```

Program 20. *people.component.ts*

```
<div *ngFor="let person of people">{{person.name}}</div>
```

Program 21. *people.component.html*

The example has two components: `AppComponent` and `PeopleComponent`. `AppComponent` contains an array of two people and the array is passed as an input for the `PeopleComponent` which then iterates the list through and prints the values of the name field for each person. `AppComponent` also contains two buttons for updating (calls `updatePeople`) and replacing (calls `replacePeople`) the array. `updatePeople` method keeps the same reference to the array and modifies the array in-place by first removing all items and then adding two new people in it. `replacePeople` assigns a completely new array to the `this.people` which means that the memory address will change. The change detection strategy for `PeopleComponent` is set to the `ChangeDetectionStrategy.OnPush`.

Now, when this code is ran in browser, initially there are two div elements with texts John and Jane. If the update button is clicked, nothing happens. The reason for that is the `OnPush` change detection strategy which makes the change detector only check if the array's (object's) reference is the same as last which it is as no new instance was assigned. In case the replace button is clicked, the view updates to contain the two new names (David and Taylor) as the array (object) now contains a new instance and thus the change is detected.

If `PeopleComponent` had any children components the change detection would be skipped for these too. If the components are again modelled as a tree structure, this means that the subtree starting from `PeopleComponent` will be skipped in change detection.

3.5 Optimizing Change Detection with Immutable Data Structures

As Angular's change detection is ran every time there is even a possibility of change it can be quite heavy performance-wise even though the change detection checks are extremely efficient already by default. To better understand the scale of change detection and the chance of optimizations let us consider a theoretical example of a large board game where there is a board of size of NxM (N rows times M columns). This board is then divided as Angular components so that each row is its own component and those components include the cells as separate components.

The `BoardComponent` acts as a root component for the board which then contains N rows (`RowComponent`) each containing M cells (`CellComponent`). This means that every time the `BoardComponent` is checked for changes, it will trigger change detection for each `RowComponent` thus triggering it for each `CellComponent`. This means that the change detection is performed in total for 1+N+NM components (`BoardComponent` + N `RowComponents` + NM `CellComponents`). So, for a board of the given size (100 x 100) this would mean 10101 components need to be checked on each change detection cycle.

If the `RowComponents` have their change detection strategy set to the `OnPush` the `Cell-Components'` change detection could be skipped in case the `RowComponents` inputs have not been changed. This alone would already mean that if the rows have not changed the amount of checks needed would be 1+N thus preventing the majority of the checks. If this approach is taken even further and the `BoardComponent` itself takes the board array (of rows containing the cells) as an input and is using `OnPush` the checks can be reduced to only one per iteration.

4. REFERENCE APPLICATION

In this chapter a reference application is introduced which will then be optimized using immutable data structures and `OnPush` change detection strategy. The application under study generates a table of a given size, and then iterates it through row by row and colors each cell within the row with a different color. Image 5 shows the visualization. Application responds on two paths, one for original un-optimized version and one for immutable optimized implementation. The first one is located in root path, whereas the latter can be found under `/optimized`. The common parts used by both implementations will be first gone through and then the differences between the implementations will be looked into. The performance analysis of the implementations is done in the next chapter.

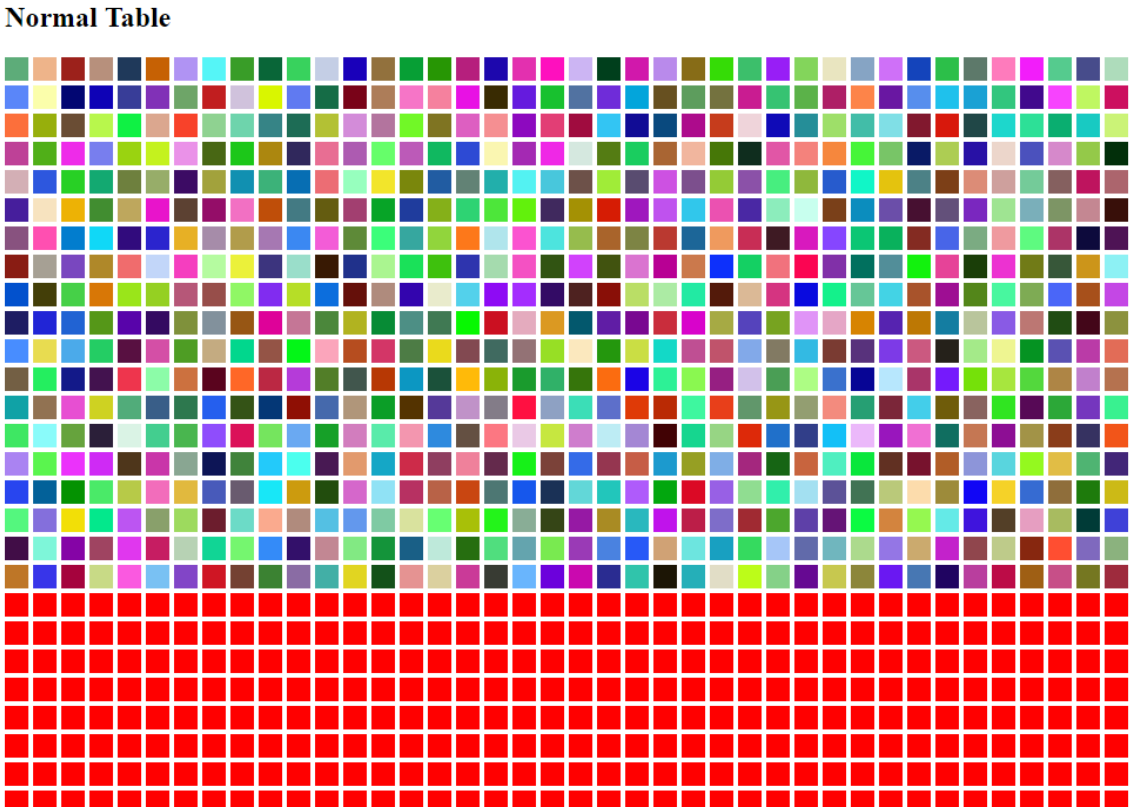

Image 5. Reference application.

The reference application is using Angular version 6.0.0 as it was the latest one at the time of writing of this thesis. It does not have many parts that would be different had earlier version be used. Also, the optimizations used would work the same way on any other version starting from the very first Angular release (2.0.0). The application and the components and services used were generated using Angular CLI version 6.0.0.

4.1 Common Parts

The application contains two Angular services that are shared within the two table implementations: `TimerService` and `RandomizerService`. The `TimerService` is more important for the performance analysis. The `RandomizerService` is only used to abstract away the logic needed to generate the colors needed. These two will next be gone shortly through.

4.1.1 TimerService

`TimerService` is used by the both implementations to track the change detection statistics. It consists of four methods: `viewChecked`, `itemChecked`, `start` and `stop`. The code can be seen in Program 22.

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

@Injectable({
  providedIn: 'root'
})
export class TimerService {
  startTime: number;
  viewChecks: number;
  itemChecks: number;

  viewChecked() {
    this.viewChecks++;
  }

  itemChecked() {
    this.itemChecks++;
  }

  start() {
    this.startTime = +new Date();
    this.viewChecks = 0;
    this.itemChecks = 0;
  }

  stop(isOnPush: boolean = false) {
    // isOnPush is needed because when using OnPush CD strategy, the initial
CD run isn't triggering anything
    const duration = +new Date() - this.startTime;
    const fps = this.viewChecks / (duration / 1000);
    const viewChecks = isOnPush ? this.viewChecks - 1 : this.viewChecks;
    const checksPerIteration = this.itemChecks / viewChecks;
    alert(`FPS: ${fps}
Checks per iteration: ${checksPerIteration} (${this.itemChecks} / ${view-
Checks})
Total duration: ${duration}ms
`);
  }
}
```

Program 22.

The `viewChecked` and `itemChecked` methods simply increment their relative counters by one. `viewChecked` is meant to be called on every change detection run, whereas the `itemChecked` should be triggered for each item component that was checked within the change detection iteration.

The `start` and `stop` methods are used to measure the time taken by the updates. `stop` also uses browsers `alert` method to show the statistics gathered containing FPS (Frames Per Second), checks performed per iteration and total time used. `stop` method also takes a parameter called `isOnPush` which determines whether the `viewChecks` needs to be adjusted or not. It is meant to be set to true if `OnPush` change detection strategy is used since when it is used, the initial view check does not trigger any item components checks and thus it would lead to an incorrect calculation.

4.1.2 RandomizerService

`RandomizerService` only provides a single method named `getColor` which simply returns a random color as a string. As mentioned already earlier, it is only needed to abstract away the logic needed to generate the random colors to be rendered on the screen to cause changes in the DOM. Implementation is presented as Program 23.

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

@Injectable({
  providedIn: 'root'
})
export class RandomizerService {
  getColor() {
    const letters = '0123456789ABCDEF'.split('');
    let color = '#';
    for (let i = 0; i < 6; i++) {
      color += letters[Math.floor(Math.random() * 16)];
    }
    return color;
  }
}
```

Program 23.

4.2 Original Application

The original part of application contains three components: `TableComponent`, `TableRowComponent` and `TableItemComponent`. `TableComponent` is the component responsible for initializing the data of the table and composing the rows from the data. Each row is represented by `TableRowComponent` which in turn holds the cells. Each cell is again represented by `TableItemComponent`. The structure is visualized in Image 6.

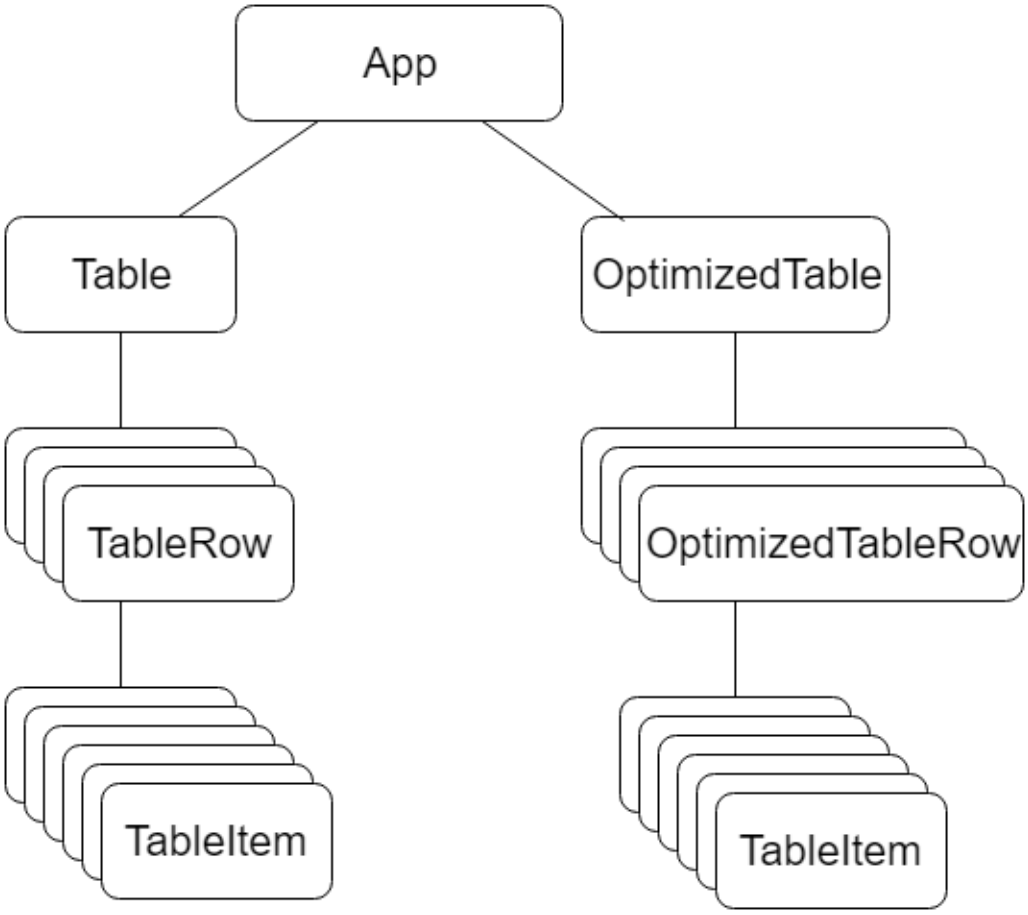

Image 6. Reference application structure.

Next each component is covered more in-depth.

4.2.1 TableComponent

`TableComponent` is the root component that is initialized when the root path is navigated to. It is responsible for initializing and updating all of the data in the table. The component's code is shown below in Program 24.

```
import { AfterContentChecked, Component, NgZone, OnInit } from '@angular/core';
import { TimerService } from '../timer.service';
import { RandomizerService } from '../randomizer.service';

@Component({
  selector: 'app-table',
  templateUrl: './table.component.html'
})
export class TableComponent implements OnInit, AfterContentChecked {
  private static rowCount = 100;
  private static rowLength = 100;
  private static iterations = 10;
```

```
  private currentRow = 0;
  private iterationCounter = 0;
  rows: any[] = [];

  constructor(private timer: TimerService,
              private randomizer: RandomizerService,
              private zone: NgZone) {
  }

  ngOnInit() {
    for (let i = 0; i < TableComponent.rowCount; ++i) {
      const arr = [];
      for (let j = 0; j < TableComponent.rowLength; ++j) {
        arr.push('red');
      }
      this.rows.push(arr);
    }
    setTimeout(() => { // Wait for the initial rendering to happen first
      this.timer.start();
    }, 0);
  }

  ngAfterContentChecked() {
    this.timer.viewChecked();
    this.updateTable();
  }

  updateTable() {
    const row = this.rows[this.currentRow++];
    if (this.currentRow <= TableComponent.rowCount) {
      setTimeout(() => {
        for (let i = 0; i < row.length; ++i) {
          row[i] = this.randomizer.getColor();
        }
      }, 0);
    } else if (this.iterationCounter < TableComponent.iterations) {
      this.currentRow = 0;
      this.iterationCounter++;
      if (this.iterationCounter < TableComponent.iterations) {
        this.updateTable();
      }
    } else {
      this.zone.runOutsideAngular(() => { // setTimeout would trigger CD with-
out this
        setTimeout(() => { // setTimeout needed to prevent too early calling
          this.timer.stop();
        }, 0);
      });
    }
  }
}
```

Program 24.

The component has three static properties called `rowCount`, `rowLength` and `iterations` to set the size of the table generated. It also gets the singleton instances of `TimerService` and `RandomizerService` via Angular's dependency injection along with

reference to Angular's `NgZone`. It implements Angular's `ngOnInit` lifecycle hook (and thus the `OnInit` interface) which is called after the component creation. In this method the data structure is initialized. The data structure is created based on the static properties mentioned earlier and each cell is initialized to the value of red. After the table is initialized, the start method of `TimerService` is called to start timing for the performance analysis. The call for `setTimeout` is required to give Angular chance to first render the red table on the screen before starting the timing.

Once everything is initialized, the Angular's change detection kicks in and triggers another lifecycle hook, called `ngAfterContentChecked` (`AfterContentChecked` interface), that has been registered. This method is called every time a change is detected in the data of the component. Within this method the `viewChecked` method of `TimerService` is called to register the change detection cycle, and after that another component method called `updateTable` is invoked. What `updateTable` does is that it takes the next row of the table and updates all of the cells in it with new color (gotten from `RandomizerService`) for each.

The actual updating is wrapped with a `setTimeout` with a delay of `0` to make it an asynchronous action. If the call was synchronous the update process would stall after first row since the changes happened within the call to `ngAfterContentChecked` would not trigger another call to itself. So the usage of the `setTimeout` queues the updating to be executed after the `ngAfterContentChecked` has already been executed and thus it will be triggered again after each update.

There are also conditionals to only do the updating until the whole table has been updated iterations times. After that the stop method of `TimerService` is called to stop the timing and to print the statistics about the performance. Here the call is also wrapped within `setTimeout` to ensure it will only be executed after the last line's update has been executed. But as discussed in previous paragraph, the usage of `setTimeout` triggers Angular's change detection. To prevent this, the call must be wrapped again to `NgZone` `runOutsideAngular` method which will run the given code outside of the Angular's change detection and this way will not trigger the change detection.

The template of the component (shown below in Program 25) is relatively simple. Besides the title it only contains one selector (`app-table-row`) which is repeated for each row in the `rows` array. The row's data itself is passed as the `colors` input property.

```
<h2>Normal Table</h2>
<app-table-row *ngFor="let row of rows" [colors]="row"></app-table-row>
```

Program 25.

4.2.2 TableRowComponent

As previously seen, single `TableRowComponent` is initialized for each row in the array. The component is extremely simple, as seen below in Program 26.

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

@Component({
  selector: 'app-table-row',
  templateUrl: './table-row.component.html'
})
export class TableRowComponent {
  @Input() colors: any;
}
```

Program 26.

Only thing to notice about the component is the input property `colors` which is passed by `TableComponent` for each `TableRowComponent`. The `colors` property array is then iterated through in the template (shown in Program 27).

```
<div>
  <app-table-item *ngFor="let color of colors" [color]="color"></app-table-
item>
</div>
```

Program 27.

The rows template contains `app-table-item` selector repeated over each color in `colors` array. The template is also wrapped within `div` element for layout purposes.

4.2.3 TableItemComponent

The `TableItemComponent` is responsible for rendering the actual color provided for it by the `TableRowComponent`. It is as simple as the `TableRowComponent` except for the lifecycle hook `ngAfterViewChecked` (`AfterViewChecked` interface) as seen below Program 28.

```
import { AfterViewChecked, Component, Input } from '@angular/core';
import { TimerService } from '../timer.service';

@Component({
  selector: 'app-table-item',
  templateUrl: './table-item.component.html'
})
export class TableItemComponent implements AfterViewChecked {
  @Input() color: string;

  constructor(private timer: TimerService) {
  }

  ngAfterViewChecked() {
```

```
    this.timer.itemChecked();
  }
}
```

Program 28.

As seen here, the component takes reference to an instance of `TimerService` via Angular's dependency injection. Then after the view has been checked and the `ngAfterViewChecked` has been called, the `itemChecked` method is called for it to register the check for the bindings of this component. Doing this makes it possible to track the amount of checks needed by the change detection.

4.3 Optimizations

The implementation with immutable data structures can be found under the path `/optimized`. On the code side it is implemented with usage of `OptimizedTableComponent` and `OptimizedTableRowComponent`. The same `TableItemComponent` is used to render the single items.

4.3.1 OptimizedTableComponent

`OptimizedTableComponent` has the same structure as the `TableComponent`. There are a few differences in the implementation, though. The differences relate to the usage of Immutable.js. As seen in the Program 29, the creation of table data includes creation of an Immutable.js `List` which is an immutable data structure for an array. These lists are then pushed to the `rows` array. Also, while updating a new list needs to be generated each time as the lists are immutable.

```
import { AfterContentChecked, Component, NgZone, OnInit } from '@angular/core';
import { TimerService } from '../timer.service';
import { RandomizerService } from '../randomizer.service';
import * as Immutable from 'immutable';

@Component({
  selector: 'app-optimized-table',
  templateUrl: './optimized-table.component.html'
})
export class OptimizedTableComponent implements OnInit, AfterContentChecked {
  private static rowCount = 100;
  private static rowLength = 100;
  private static iterations = 10;

  private currentRow = 0;
  private iterationCounter = 0;
  rows: any[] = [];

  constructor(private timer: TimerService,
              private randomizer: RandomizerService,
              private zone: NgZone) {
  }
```

```
  ngOnInit() {
    for (let i = 0; i < OptimizedTableComponent.rowCount; ++i) {
      const arr = [];
      for (let j = 0; j < OptimizedTableComponent.rowLength; ++j) {
        arr.push('red');
      }
      this.rows.push(Immutable.List(arr));
    }
    setTimeout(() => { // Wait for the initial rendering to happen first
      this.timer.start();
    }, 0);
  }

  ngAfterContentChecked() {
    this.timer.viewChecked();
    this.updateTable();
  }

  updateTable() {
    if (this.currentRow < OptimizedTableComponent.rowCount) {
      setTimeout(() => {
        const arr = [];
        for (let i = 0; i < OptimizedTableComponent.rowLength; ++i) {
          arr[i] = this.randomizer.getColor();
        }
        this.rows[this.currentRow++] = Immutable.List(arr);
      }, 0);
    } else if (this.iterationCounter < OptimizedTableComponent.iterations) {
      this.currentRow = 0;
      this.iterationCounter++;
      if (this.iterationCounter < OptimizedTableComponent.iterations) {
        this.updateTable();
      }
    } else {
      this.zone.runOutsideAngular(() => { // setTimeout would trigger CD with-
out this
        setTimeout(() => { // setTimeout needed to prevent too early calling
          this.timer.stop(true);
        }, 0);
      });
    }
  }
}
```

Program 29.

Program 30 shows the corresponding template.

```
<h2>Optimized Table</h2>
<app-optimized-table-row *ngFor="let row of rows" [colors]="row"></app-opti-
mized-table-row>
```

Program 30.

4.3.2 OptimizedTableRowComponent

The `OptimizedTableRowComponent` is equivalent to the `TableRowComponent` except for the change detection strategy `OnPush` been set for it. As discussed earlier in chapter 3, the usage of `OnPush` change detection strategy means that the component's sub-tree of other components can be totally skipped in case the inputs of this component still reference the same object. This is the case in this application since only one row is updated per iteration and thus all the rest of the rows can be skipped. Program 31 shows the code for the component.

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

@Component({
  selector: 'app-optimized-table-row',
  templateUrl: './optimized-table-row.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class OptimizedTableRowComponent {
  @Input() colors: any[];
}
```

Program 31.

The corresponding template is shown as Program 32.

```
<div>
  <app-table-item *ngFor="let color of colors" [color]="color"></app-table-
item>
</div>
```

Program 32.

5. PERFORMANCE ANALYSIS

The reference application contains two implementations for the same task of rendering a table row by row. The first implementation is done the traditional way while the second one is using immutable lists from Immutable.js library. The hypothesis of this master's thesis is that the usage of immutable data structures can greatly improve the performance of the change detection by reducing the checks needed on each change detection run. As the number of checks necessary is reduced, the change detection takes less time and thus the overall performance increases. This performance was measured with the reference application and the results are shown in this chapter.

5.1 Measurement

The reference application tracks the average FPS (frames per second) and the count of checks for item components per change detection iteration during the table update. From these the FPS is used as the main measure for performance as it describes the change detection efficiency as seen by user. The count of checks for item components provides an explanation for the performance difference since lesser checks lead to better overall performance.

As the logic and implementations of both approaches are as close to each other as possible, and the performance monitoring is done by common service, the results are comparable. The measurements shown next were ran ten times per approach to reduce the effect of possible random interfering matters.

5.2 Results

The raw results along with the average FPS of the ten measurements for both approaches are shown in the Table 1. The numbers are represented with 5 decimals as that is determined to provide enough accuracy. The results are of course specific to the machine they were collected on but are still generalizable since the relative results should remain about the same on every machine.

Measurement	Original	Optimized
1	53.02702	64.05632
2	52.75262	64.83717
3	52.42857	65.43834
4	53.30687	64.10107
5	51.99282	63.78911
6	49.27497	64.07869
7	52.90471	61.43634
8	53.09863	62.02118
9	53.19612	63.83719
10	51.86057	63.64898
Average	52.38429	63.72443

Table 1.

Based on these averages, the approach using immutable data structures reaches 1.21648 times bigger FPS than the original approach. The reason for this can be found on the items checks per change detection iteration row. The count of components checked per each change detection cycle were 10 000 for original approach and 100 for optimized one. Thus, the usage of the immutable data structures reduces the need for checks by 100 times in this setup.

6. DISCUSSION & CONCLUSIONS

The whole field of the frontend development is currently in a big turning point. The traditional flaky constructions on top of legacy JavaScript with usage of global variables and no modules are becoming part of history. This is a great direction and the frontend nowadays is much closer to the coding for any seriously taken platform than it was even just a few years ago. This is extremely important as the web is becoming more and more important all the time compared to the native applications.

The applications built on top of the browsers are already so complex that managing them without the well-suited tools would be disastrous. Luckily new tools have emerged to solve these problems. New, improved tooling is popping up in all of the sectors of the development all the time. Some even argue they pop up too often to keep track of all of the advancements. Consider for example static typing (TypeScript & Flow), build tooling (Webpack, Parcel, Bazel and others) and of course frameworks (Angular, React and Vue.js among them) to mention a few areas that has only been launched on this decade and are already widely adopted. These make the lives of the developers a lot easier. But this all comes with a cost. As web is evolving faster than any other ecosystem ever has, it has proven to be extremely hard to keep up with all the latest trends in tooling, libraries and frameworks.

Traditionally two of the most important factors on choosing a framework are the bundle size and performance. The bundle size is not that much of a problem for the new frameworks anymore as all frameworks are designed considering the size. Also, the networks are blazing fast nowadays and getting even faster all the time. Thus, the interest turns into the performance. Performance-wise the implementation details of different frameworks are quite a bit different. Whereas React relies on virtual DOM approach, Angular does utilize the dirty checking made famous by its predecessor Angular.js. It does this much more intelligently than Angular.js and thus the performance is ten folds better. Still, there are optimizations available for the developer to take it even further, if necessary.

Optimizing Angular's change detection with usage of immutable data structures and `OnPush` strategy is an example of such an approach. With it, an optimally constructed component composition can gain huge performance improvements without the need for large refactoring of the application logic. It does neither compromise the readability or the maintainability. Actually, it does quite the contrary by providing a clear indication of what should be immutable and what mutable in the data model of an application.

The measurements for the reference application clearly show that there is a performance impact from the usage of the immutable data structures along with the `OnPush` change

detection strategy. Yet, the impact is not as big as one would expect based on the difference in amount of checks needed. Whereas the unoptimized version does 10 000 checks on each iteration, the optimized version only needs to perform 100 checks. Thus, the optimized version requires 100 times less checks than the unoptimized one.

While this difference in checks required on each iteration seems huge, it is only the first part of the change detection process. After the changes have been detected they need to be updated to the DOM. The DOM operations and the changes they cause in the user interface are extremely heavy performance-wise. These operations are executed by the very same thread as where the updates to our data structure are done.

This leads to a situation where the overall performance of the application is based not only on the performance of the Angular's change detection checks but also on the efficiency of the DOM updates made by Angular and rendering speed of the browser. As the latter parts of the process are more performance-heavy, the 100-fold reduce in number of checks does not mean the application as whole will perform 100 times better. Yet, it can clearly be seen based on the measurements that also the application's overall performance is significantly better when the optimization is enabled.

While the time taken by the change detection checks is relatively stable and deterministic by nature, the same can not be said about the browser's performance at any given time. For example, JavaScript's garbage collection process and optimizations applicable for each re-rendering cycle affect the performance in a very indeterministic way. This can also be seen from measurements as there is some deviation in the results for both unoptimized and optimized versions.

The performance gains in the case study of this thesis are of course heavily dependent of the setup on hand. Yet, it still proves the usability of this technique to further optimize the already well-performing Angular change detection system. Improvement of 1.21648-fold for this number of components boosts also the visual result on the browser heavily whereas the original rendering is a little sticky.

The usage of the immutable data structures is still quite clumsy in JavaScript. This is mostly because of the lack of the native support by the ECMAScript standards. Libraries such as the Immutable.js do good work to fulfill this gap but these approaches are still limited by the language features. Maybe the immutable data structures will be seen in the ECMAScript standard someday but at the moment it does not seem likely as there is not even a proposal for them in the TC39 committee's list of proposals (TC39 proposals).

SOURCES

AJAX. Cited 18.5.2018. Available: https://developer.mozilla.org/en-US/docs/Web/Guide/AJAX

Angular. Cited 17.5.2018. Available: https://angular.io/

Angular.js first releases. Cited 6.5.2018. Available: https://github.com/angular/angular.js/releases?after=v0.9.4

Angular 6 release notes. Cited 6.5.2018. Available: https://blog.angular.io/version-6-of-angular-now-available-cc56b0efa7a4

Angular lifecycle hooks. Cited 6.5.2018. Available: https://angular.io/guide/lifecycle-hooks

Angular ViewRef code. Cited 6.5.2018. Available: https://github.com/angular/angular/blob/0cb4f12a7a57087ec4e8329a04d5dfc430764b45/packages/core/src/linker/view_ref.ts#L31

Babel. Cited 6.5.2018. Available: https://babeljs.io/

Davis, Adam L. Functional Programming - Learning Groovy. 2016. ISBN 978-1484221167.

Dart. Cited 6.5.2018. Available: https://www.dartlang.org/

DOM specification. Cited 6.5.2018. Available: https://www.w3.org/DOM/DOMTR

Driscoll, J. R., Sarnak, N., Sleator, D. D., Tarjan, R. E. Making data structures persistent. STOC '86 Proceedings of the eighteenth annual ACM symposium on Theory of computing p. 109-121.

Eich, Brendan - CEO of Brave. Fin JS. 2016. Cited 6.5.2018. Available: https://www.youtube.com/watch?v=XOmhtfTrRxc&feature=youtu.be&t=2m5s

Haq, Zia Ul, Khan, Gul Faraz, Hussain, Tazar. A Comprehensive analysis of XML and JSON web technologies. 2015. King Saud University, Saudi Arabia.

Koretskyi, Maxim. Everything you need to know about change detection in Angular. 2017. Cited 6.5.2018. Available: https://blog.angularindepth.com/everything-you-need-to-know-about-change-detection-in-angular-8006c51d206f

Kulkarni, Rohit, Chavan, Aditi, Hardikar, Abhinav. Transpiler and it's Advantages. (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 6 (2), 2015, 1629-1631

RFC 7159. JSON Specification. Cited 18.5.2018. Available: https://tools.ietf.org/html/rfc7159

Latest ECMAScript standard. Cited 6.5.2018. Available: https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

Lee, Hongki, Won, Sooncheol, Jin, Joonho, Cho, Junhee, Ryu, Sukyoung. SAFE: Formal Specification and Implementation of a Scalable Analysis Framework for ECMAScript. p. 96. FOOL 2012 : 19th International Workshop on Foundations of Object-Oriented Languages.

Leroy, Xavier. The effectiveness of type-based unboxing. TIC 1997: Workshop Types in Compilation, 1997, Amsterdam, Netherlands.

Mikowski, Michael S., Powell, Josh C. Single Page Web Applications. 2014. Manning Publications. p. 5. ISBN 978-1-617290-75-6.

Nagele, Thomas. Client-side performance profiling of JavaScript for web applications. 2015. Master's thesis. Radboud University Nijmegen.

Netscape. Netscape and Sun announce JavaScript, the open, cross-platform object scripting language for enterprise networks and the Internet. 1995. Cited 6.5.2018. Available: https://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html

Nurseitov, Nurzhan, Paulson, Michael, Reynolds, Randall, Izurieta, Clemente. Comparison of JSON and XML Data Interchange Formats: A Case Study. 2010. Montana State University.

Okasaki, Chris. Purely Functional Data Structures. Cambridge University Press. 1998. ISBN 0-521-66350-4.

Parviainen, Tero. Change and Its Detection in JavaScript Frameworks. 2015. Cited 6.5.2018. Available: https://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html

Ramos, Miguel, Valente, Marco Tulio, Terra, Ricardo. AngularJS Performance: A Survey Study. 2017. IEEE Software (Volume: 35, Issue: 2, March/April 2018). DOI: 10.1109/MS.2017.265100610. p. 72-79. 2017. ISSN 0740-7459.

Rauschmayer, Axel. JavaScript's type system. 2013. Cited 18.5.2018. Available: http://2ality.com/2013/09/types.html

Stefanov, Stoyan. JavaScript Patterns. 2010. O'Reilly Media, Inc. p. 5. ISBN 9781449396947.

Strassner, Tom. XML vs JSON. Cited 6.5.2018. Available: http://www.cs.tufts.edu/comp/150IDS/final_papers/tstras01.1/FinalReport/FinalReport.html

TC39 committee. Cited 6.5.2018. Available: https://www.ecma-international.org/memento/TC39.htm

TC39 proposals. Cited 6.5.2018. Available: https://github.com/tc39/proposals

TypeScript recommendation. Cited 6.5.2018. Available: https://angular.io/guide/typescript-configuration

Web APIs. Cited 6.5.2018. Available: https://developer.mozilla.org/en-US/docs/Web/API

XHR standard. Cited 6.5.2018. Available: https://xhr.spec.whatwg.org/

XML. Cited 6.5.2018. Available: https://www.w3.org/TR/xml/

Zone.js. Cited 6.5.2018. Available: https://github.com/angular/zone.js/