Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of Information Engineering

Diploma Thesis

Type Checker for Typescript Application

Bsc. Sushmita Parajuli

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

DIPLOMA THESIS ASSIGNMENT

BSc. Sushmita Parajuli, BSc

Systems Engineering and Informatics
Informatics

Thesis title

Type checker for TypeScript application

Objectives of thesis

The main objective of this thesis is to develop and evaluate a script for type checking of JavaScript
libraries to improve the development of TypeScript application.

Partial objectives:
• Characterize the advantages of TypeScript over JavaScript.
• Build type checker script.
• Analyze the type errors returned by different used JavaScript libraries.

Methodology

At the beginning, depth research on the topic will be done to solve the first partial objective. As a practical
part, automated type checker script will be developed to test different packages used in an experimental
web application. The automated type test script will be used to find different type related bugs and errors
returned by used libraries. The script will be evaluated using selected metrices (performance, reusability,
and maintainability) by testing the application with and without the script.

Based on the synthesis of theoretical knowledge and the results of the practical part, conclusions of the
work will be formulated.

The proposed extent of the thesis

60 – 80 pages

Keywords

TypeScript, React.js, Node.js, JavaScript, JSX, error handling, automated test case.

Recommended information sources

Ball, T., de Halleux, P., & Moskal, M. Static TypeScript: an implementation of a static compiler for the TypeScript language. In Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes(2019). DOI 10.1145/3357390.3361032

Bierman, G., Abadi, M., & Torgersen, M. Understanding typescript. In European Conference on Object-Oriented Programming (2014).DOI 10.1007/978-3-662-44202-9_11

HICKEY, James. Refactoring TypeScript. Packt Publishing Ltd, 2020. ISBN 9781839218040.

Kristensen, E. K., & Møller, A. Inference and evolution of typescript declaration files. In International Conference on Fundamental Approaches to Software Engineering(2017). DOI 10.1007/978-3-662-54494-5_6.

ROZENTALS, Nathan. Mastering TypeScript. Packt Publishing Ltd, 2021. ISBN 9781800564732.

Expected date of thesis defence

2021/22 SS – FEM

The Diploma Thesis Supervisor

Ing. Jan Masner, Ph.D.

Supervising department

Department of Information Technologies

Electronic approval: 16. 8. 2021

doc. Ing. Jiří Vaněk, Ph.D.

Head of department

Electronic approval: 19. 10. 2021

Ing. Martin Pelikán, Ph.D.

Dean

Prague on 16. 03. 2022

Declaration

I declare that I have worked on my diploma thesis titled "Type checker for TypeScript Application" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the diploma thesis, I declare that the thesis does not break any copyrights.

In Prague on 30-03-2022   _____

Acknowledgment

I would like to thank Ing. Jan Masner, Ph.D. for his advice and support during my work on this thesis and my family for motivating me to finish my thesis on time.

TypeChecker for TypeScript Application

Abstract

Developers are starting to use Typescript, a typed dialect of JavaScript, to develop huge and complicated apps. JavaScript libraries are integrated into Typescript applications using declaration files, which are a typed description of their APIs. The standard public repository for these files is DefinitelyTyped. Volunteers manually populate and maintain the repository, which is error-prone and time-consuming. Inconsistencies between a declaration file and javascript implementation cause the Typescript IDE to give inaccurate feedback, which leads to inappropriate use of the underlying JavaScript library.

This thesis presents typechecker, a script that assigns and checks for the types that are required by JavaScript libraries or static files. To evaluate the script, 2 JavaScript libraries and some API responses were run against the script.

It has been found that some of the libraries are ignoring type related errors intentionally. As the demand of TypeScript on JavaScript application are increasing, it has been found that typechecker is beneficial for libraries which are not maintained frequently. The result of the thesis can be used for generating type guards, which verifies the types of data.

Keywords: TypeScript, React.js, Node.js, JavaScript, JSX, error handling, Types, Declaration Files

Typechecker pro aplikaci TypeScript

Abstrakt

Vývojáři začínají používat Typescript, typizovaný dialekt JavaScriptu, k vývoji obrovských a komplikovaných aplikací. Knihovny JavaScriptu jsou integrovány do aplikací Typescript pomocí deklaračních souborů, které jsou strojovým popisem jejich API. Standardní veřejné úložiště pro tyto soubory je rozhoděněTyped. Dobrovolníci ručně naplňují a udržují úložiště, což je náchylné k chybám a časově náročné. Nesrovnalosti mezi deklaračním souborem a implementací javascriptu způsobují, že Typescript IDE poskytuje nepřesnou zpětnou vazbu, což vede k nevhodnému použití základní knihovny JavaScriptu.

Tato práce představuje typecker, skript, který přiřazuje a kontroluje typy, které jsou vyžadovány knihovnami JavaScriptu nebo statickými soubory. Pro vyhodnocení skriptu byly proti skriptu spuštěny 2 JavaScriptové knihovny a některé odpovědi API.

Bylo zjištěno, že některé knihovny záměrně ignorují chyby související s typem. Vzhledem k tomu, že poptávka po TypeScriptu na aplikaci JavaScript roste, bylo zjištěno, že typecker je výhodný pro knihovny, které nejsou často udržovány. Výsledek práce lze použít pro generování typových strážců, které ověřují typy dat.

Klíčová slova: TypeScript, React.js, Node.js, JavaScript, JSX, zpracování chyb, typy, deklarační soubory

Table of content

List of Figures

List of Source code

List of abbreviations

API   -   Application Programming Interface
JS   -   JavaScript
JSON   -   JavaScript Object Notation
ECMA -   European Computer Manufacturer's Association
LTS   -   Long Term Support
OOP   -   Object-Oriented Programming
UI   -   User Interface
NPM   -   Node Package Manager
IDE   -   Integrated Development Environment

1 Introduction

In this technological era, there is high demand for complex and secured web applications. There are numerous technologies available that can satisfy the demand of complex web applications. Selecting the best framework and libraries which are understandable, manageable and predictable is tough. Over the past few years, optionally typed languages are mostly preferred by developers. According to the survey done by stackoverflow [12] under the category of most loved technology, 67.1% of developers are working on TypeScript and are willing to contribute further whereas 66.7% of developers are working on Python and 58.3% of developers are working on JavaScript.

The optionally typed languages allows programmers to flexibly switch between dynamically and statically typed paradigms, resulting in increased productivity and flexibility as well as early error detection. When type checking is done at compile time rather than run time then that programming language is said to use static typing. In the case of static typing, types are associated with the variables rather than with values. Static type systems decrease development time and increase the code quality, since they restrict developers to express themselves in a desired way [13]. When the majority of type checking is done at run-time rather than compile-time, then that programming language is said to be dynamically typed or dynamic. In the case of dynamic typing, types are associated with the values rather than with variables. Programming languages' dynamic and reflective capabilities are powerful elements that programmers frequently describe as incredibly beneficial. The ability to change a program at runtime, on the other hand, can be a blessing (in terms of flexibility) and a burden (in terms of tool support) [14].

Optionally typed language provides freedom to developers to express themselves. Meanwhile it's not possible to develop applications from scratch every time. Developers need to use existing packages and libraries which provide required options and features. Typescript provides a typed API model to describe the behaviors of untyped libraries and packages. At runtime, the API models are ignored and the untyped library is executed regardless of the types in the API models. That particular API model is known as declaration files (.d.ts) in case of TypeScript. Declaration files are just TypeScript files that describe the structure of an existing JavaScript codebase. Definitely Typed is just a simple repository on GitHub that hosts TypeScript declaration files for all JavaScript packages [15].

The declaration files are written and maintained by hand, which is time consuming and prone to errors. TypeScript application programmers are impacted by mismatches between declaration files and the accompanying JavaScript implementation of libraries. The type checker generates incorrect type error messages, code navigation and auto-completion are erroneous, resulting in programming errors and higher development expenses. Many strategies for automatically detecting many kinds of errors in many different programming languages have been developed over the years in programming language research such as type systems and dynamic analysis.

2 Objectives and Methodology

2.1 Objectives

The main objective of this thesis is to develop and evaluate a script for type checking of JavaScript libraries to improve the development of TypeScript application.

Partial Objectives
- Characterize the advantages of TypeScript over JavaScript.
- Build type checker script.
- Analyze the type errors returned by different used JavaScript libraries.

2.2 Methodology

At the beginning, depth research on the topic will be done to solve the first partial objective. As a practical part, type checker script will be developed to test different packages used in an experimental web application. The type test script will be used to find different type related bugs and errors returned by used libraries. The script will be evaluated using selected metrics (performance, reusability, and maintainability) by testing the application with and without the script.

Based on the synthesis of theoretical knowledge and the results of the practical part, conclusions of the work will be formulated.

3 Literature Review

3.1 Data Types

In programming, a data type is a categorization that describes the type of value a variable possesses and the types of mathematical, relational, and logical operations that can be performed on it without creating an error. A string, for example is a data type for classifying text, whereas an integer is a data type for classifying whole integers. To effectively design event and entity characteristics, a thorough understanding of data types is essential. To assure data accuracy and prevent data loss, a well-defined tracking plan must include the data type of each property. String, integer, Boolean, array, enum, date are different data types.

A programming language is said to be strongly typed when it requires a variable to only be utilized in ways that respect its data type. A programming language is said to be weakly typed if it permits a variable of one data type to be used as if it were a value of another data type. Every variable's value has a static type in every programming language, however the type may be one whose values are categorized into one or more classes.

3.2 Type checking

The goal of type checking is to ensure that the program is type-safe, reducing the risk of type mistakes. In other words, type checking is a program analysis that confirms that the types used in the program are correct. The goal of type checking is to ensure that data is type safe. Type checking might take either at compile time or during run time.

3.2.1 Static Typing

If the type of a variable is known at compile time rather than at run time, then the language is known as statically typed language. Once a variable has been declared with a type in a statically typed language, it can never be assigned to another variable of that type and doing so will result in a type error at build time. Java, C, C++, FORTAN, PASCAL and Scala are statically-typed languages.

3.2.2 Dynamic Typing

If the type of a variable is checked during run-time, the languages is dynamically typed. Variables are assigned to objects at run-time in dynamically typed languages via assignment statements, and the same variables might be bound to objects of different types at the same time. When compared to static type checking, dynamic type checking produces less optimal code. It also contains the possibility of run-time type errors and requires that run-time checks be performed for each program execution. JavaScript, PHP, Python, Ruby, Lisp are dynamically-typed languages.

3.3 JavaScript

JavaScript was developed by Brendan Eich in 1995 for Netscape 2. In 1997 JavaScript was handed over to ECMA (European Computer Manufacturer's Association). After the handover, the Mozilla foundation continued development of JavaScript for their Firefox browser. JavaScript is a light-weight, interpreted and object-oriented scripting language with C inspired syntax.

JavaScript is a scripting language that enables us to enhance static web applications by providing dynamic, personalized and interactive content [1]. JavaScript in an interpreted language rather than compiled language because JavaScript code can't be directly converted to machine readable code. JavaScript has its own interpreter which converts JavaScript code to machine code. Conversion of code by interpreter is done during the time of code execution.

JavaScript which follows ECMASCRIPT standard was originally designed to be a Web Scripting language, providing a mechanism to enliven web pages in browsers to perform server computation as a part of web-based client-server architecture [2]. ECMAScript is being utilized for a wide range of programming activities in a variety of environments and scales, expanding beyond simple scripting. With the expansion of usage of ECMAScript, it is now considered a fully featured general-purpose programming language. JavaScript is a language that is always evolving and becoming more complicated. The number of published versions explained on [3] can demonstrate its popularity among developers. Lightweight Scripting language, dynamic typing, object-oriented programming support, functional style, platform independent, prototype-based, interpreted language, Async

processing, client-side and server side validation are the most important features supported by JavaScript.

JavaScript is a backward compatible scripting language. Babel is a JavaScript compiler or a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environment [4]. Modern ECMAScript standard functions may not be supported by older browsers. To solve this problem Polyfill is used. Polyfill is a piece of code (usually JavaScript on the web) used to provide modern functionality on older browsers that do not natively support it [5]. JavaScript is a reliable choice for web development, and many of its frameworks are based on it. In recent days, Node.js, Vue.js, Angular JS, Ember.js and React are popular frameworks preferred by JavaScript developers.

Figure 1: Survey by Redmonk for the first quarter of 2021, first position is taken by JavaScript – source: [11]
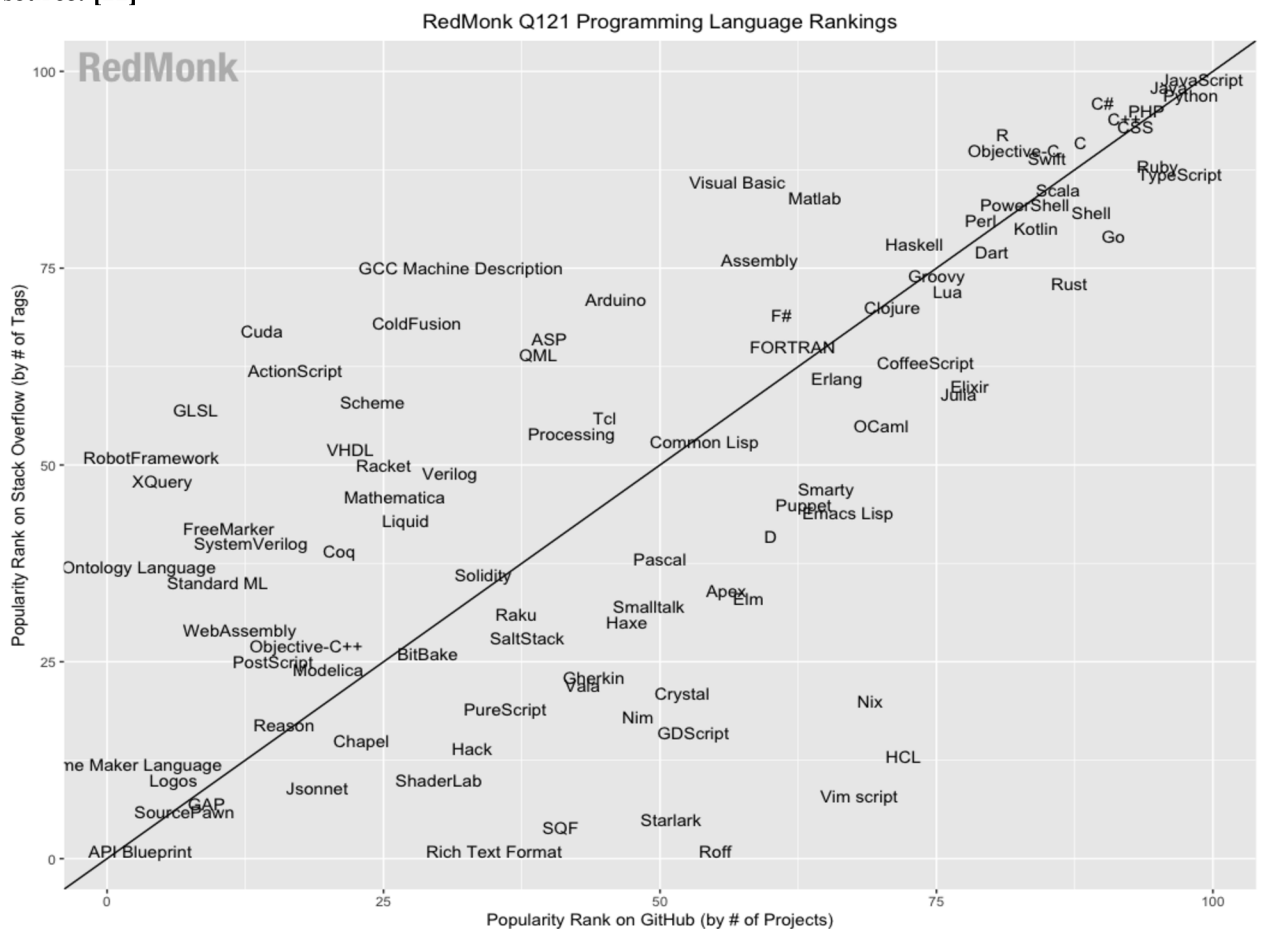
As shown in figure 1, X-axis shows the popularity rank on Github (by # of projects) whereas Y-axis shows the popularity rank on Stack Overflow (by # of Tags). As we can see, JavaScript stands on first position leaving python on second position.

3.3.1 Types in JavaScript

JavaScript is a dynamic, loosely typed language. Variables in JavaScript are not linked to any specific value type, and any variable can be assigned (and re-assigned) values of any kind. Most type checkers for statically typed languages would reject many legitimate JavaScript expressions due to the dynamic type system. Static analysis of JavaScript is more difficult than static analysis of statically typed languages due to the dynamic types. A static analysis in a statically typed language can assume the type of the expression at any program point using a type annotation. These assumptions can be used to filter values during static analysis as well as to construct conservative overestimates by only looking at the indicated types of variables [34]. Type coercion is a feature of JavaScript that allows values to be automatically converted from one type to another. Developers can often trust the language to accomplish what the developers intended because of these types of coercions.

Source code 1: Type coercisons in case of of string and integer

```javascript
const test = '20'+ 21;
console.log(test);
```

Source code 2: Type coercision in case of "=="

```javascript
21 == '21'  // true
```

When the above source code 1 gets executed, the string '2021' will be printed on the console. The number 21 is explicitly converted to string 21 resulting in the whole text as string. The other type of coercions in the case of "==" operation. In javaScript "==" is used to compare values.

when the above source code 2 gets executed, it is evaluated as true as string(21) is equal to 21because "==" compares their values ignoring its particular types. Javascript internally typecasts the string 21 to integer type and values are equal for both.

3.3.2 Objects

Objects are the most significant data type in JavaScript and serve as the foundation for modern JavaScript. These objects differ from JavaScript's primitive data-types (Number, String, Boolean, null, undefined, and symbol) in that, while they all store a single value, these objects store many values (depending on their types). With braces {.....} and an

optional list of properties, an object can be constructed. A property is a "key: value" pair, with the key being a string (sometimes known as the "property name") and the value being anything [35].

Source code 3: university as a object

```
let university = {
  name: '',
  address: '',
};
```

When the above source code 3 gets executed, the dynamic property of a university is defined.

Source code 4: New dynamic property of object university

```
university.name = 'CULS';
```

When the above source code 4 gets executed a new dynamic property of object university is created. The fact that attributes on objects are defined and then read based on arbitrarily complex computations is one of the main reasons why static analysis of JavaScript can be difficult. If a static analysis cannot determine which property of an object is being written to, it may assume that all of the object's properties have been overwritten, causing subsequent property reads on the object to be modeled imprecisely.

3.3.3   Inheritance

There is only one construct in JavaScript: objects. Each object has a private property that contains a connection to its prototype, which is another object. That prototype object has its own prototype, and so on until you reach an object with null as its prototype. Null is the final link in this prototype chain because it has no prototype by definition [36]. When an inherited function is called, "this" refers to the inheriting object rather than the prototype object where the function is its own property.

Source code 5: Inheritance

```
let obj = {
  a: 2,
  b: function () {
    return this.a + 1;
  },
};
console.log(obj.b()); // 3
// When calling obj.a in this case, 'this' refers to obj
let newObj = Object.create(obj);
// newObj is an object that inherits from obj
newObj.a = 4; // creates a property 'a' on newObj
console.log(newObj.m()); // 5
// when newObj.m is called, 'this' refers to newObj.
// So when newObj inherits the function m of o,
```

When above source code 5 gets executed inherited function is called.

3.3.4   Functions

In JavaScript, functions are one of the most basic building components. In JavaScript, a function is comparable to a procedure—a series of instructions that performs a task or calculates a value—but a process must accept some input and return an output with some evident relationship between the input and the output to qualify as a function [37]. Function overloading is not supported in JavaScript. Rather, programmers can create functions that detect the types of their inputs during execution.

Source code 6: add function

```
function add(number) {
  return number + number;
 }
```

When the above source code 6 gets executed it creates a function. The function add takes one input parameter number. The function consists of one statement that says to return the parameter of function added to itself.

3.3.5   JavaScript new features ECMAScript 2021

ECMAScript 2021 has introduced various new features, which makes developers task easy. The newly available features are replaceAll, Promise.any, AggregateError, logical

assignment operators (??=, &&=, ||=), WeakRef, FinalizationRegistry, numeric literals (1_000) and Array.prototype. sort [16].

3.3.6   JavaScript Library

The intricacy of creating new applications makes it difficult for developers to work with programming languages. As a developer, one of the aims is to decrease code and have reusable code rather than redundant code all over the place. Practitioners create libraries that reduce the complexity of web application programming while also requiring less code. A JavaScript library is a piece of code written in JavaScript with the goal of improving the practical and aesthetic aspects of web development. A library's major purpose is to make development easier and faster, to produce code in less time, and to code less [25].

3.3.7   JavaScript Framework

Libraries are usually grouped together based on their functions, or they can serve a purpose as part of a skeleton, which is referred to as framework [26]. A framework was created to save developers time when designing web applications. It is intended to speed up not only the development of the program, but also the development of online applications that must adhere to a specified JavaScript framework's structure. Other developers can maintain the code thanks to the framework. When you use a framework, you are obliged to develop in the manner that the framework dictates, therefore you must adhere to a set of rules and structure in order to comply with the framework. The majority of JavaScript frameworks are open-source, which implies that a community is always working to enhance the framework and fix flaws.

Frameworks for JavaScript include a collection of tools, functions, and high-level abstractions. Visual design, animation, drag & drop, and event management are just a few of the user-friendly features provided by frameworks. An increase in the number of open source communities has occurred from the rising use of frameworks that rely on JavaScript. Many new JavaScript frameworks have been developed in recent years, and established frameworks have been enhanced with new features [28].

3.4 React.js

React (also known as React.js or ReactJS) is a free and open-source Javascript frontend library for creating user interfaces using UI components [49]. Everything in React is a component. The most significant benefit of using components is that we can update any component at any time without affecting the rest of our apps. When used in bigger, real-time applications where data changes often, this functionality is most useful. ReactJs immediately updates the relevant component whose state has changed whenever any data is added or altered. This eliminates the need for the browser to reload the entire program to reflect the changes.

3.4.1 JSX

JavaScript XML is abbreviated as JSX. It's an XML/HTML like syntax that React uses. It enhances ECMAScript to allow for the coexistence of XML/HTML – like text and JavaScript react code. Pre-processors like Babel employs this syntax to convert HTML like constent present in JavaScript files into regular JavaScript objects. We may take it a step further with JSX by inserting the HTML code inside the javaScript once more [50]. This simplifies HTML coding and improves JavaScript speed while making our application more robust.

Figure 2: JSX of React - source:[50]

Figure 2 shows, combination of JavaScript with HTMl makes JSX of ReactJs.

3.4.2 Virtual DOM

Virtual DOM, like an actual DOM, is a node tree that lists items, attributes and content as Objects with properties. The render method in React produces a node tree from React

components. The tree is then updated in response to data model mutations induced by various actions taken by the user or by the system [51].

Figure 3: Virtual DOM - source:[51]
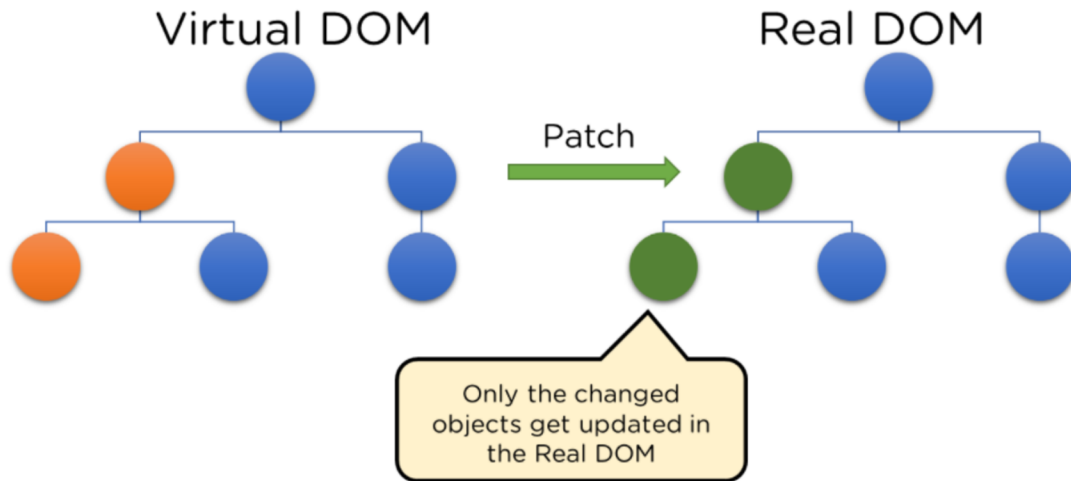
Figure 3 shows how Virtual DOM and Real DOM interacts with each other.

3.4.3 Testability

State functions can be implemented using React views (state is an object which determines how a component will render and behave). As a result, we can easily alter state of the components we supply to the ReactJs view and inspect the output as well as the triggered actions, events, functions and so on. This makes it simple to test and troubleshoot React apps.

3.4.4 Server-Side Rendering

Server-side rendering enables us to render the initial state of our rect components only on the server side. With SSR, the server's response to the browser is reduced to the page's HTML, which is now ready to be rendered. As a result, the browser can begin rendering without needing to wait for all of the JavaScript to load and run. As a result, the web page loads more quickly. Despite the fact that React is still download JavaScript, constructing the virtual DOM, linking events, and so on at the back end, the user will be able to see the web page [50].

3.4.5   Data Binding

ReactJs, unlike other frameworks, uses one-way data binding or unidirectional data flow. The main benefit of One-Way data binding is that data flows in a single path across the program, giving us more control. As a result, the applicaion's state is stored in particular stores, leaving the rest of the components loosely connected. This increases the flexibility of our application, resulting in enhanced efficiency [51].

3.5   TypeScript

TypeScript is an object-oriented programming language developed and maintained by Microsoft. Typescript is a scripting language that is a superset of JavaScript. In other words, Typescript code is converted to equivalent JavaScript code before execution. Typescript adopts its basic language feature from EcmaScript5 specification i.e. official specification of [6]. Compilation, Strong static typing, type definitions and object-oriented programming are the features supported by Typescript. Typescript checks for an error before execution and does so based on the kinds of its values. Language, the Typescript compiler and the Typescript language service are basic three components of Typescript. One of the Typescript's core principles is that type checking of typescript focuses on the shape that values have.

After 2 years of internal development lead by Anders Hejlsberg at Microsoft, Typescript was first made public in the year 2012. JavaScript was unable to solve problems of large scale-applications among both Microsoft and their internal customers. As a result, Typescript was developed to solve complex problems of JavaScript. The first release of Typescript was typescript version 0.8 in October 2018. With the increase in use and requirements of typescript in large-scale applications, typescript continuously improved its features and performance on later release. Altogether, there are 1,874 total versions (dev, beta, LTS) of typescript till date (2021). Version 4.2.2 is the latest available version of Typescript.

Figure 4: Conversion of TypeScript code to Javascript - source:[52]
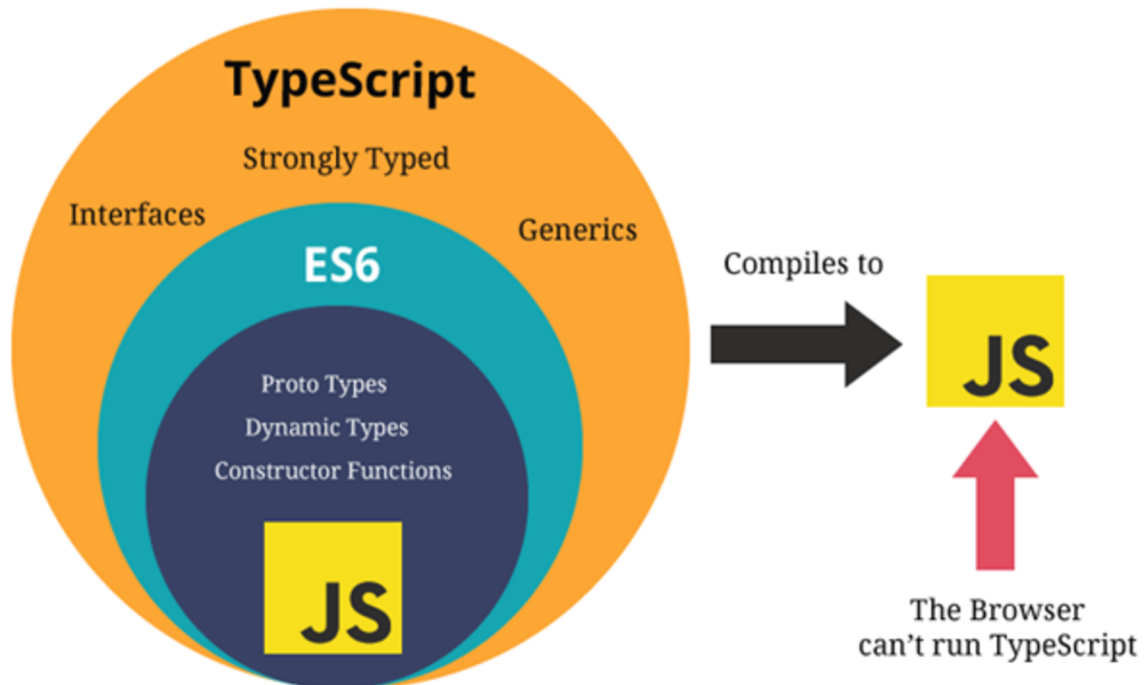
Above figure 4 shows how typescript files get compiled to Javascript.

In recent years, Typescript has gained popularity among JavaScript as well as backend developers. Improved maintainability, code consistency, and future browser support are the main reasons for people's attraction [7]. Typescript provides mode features to make the codebase easy to write and understand. Implementing Typescript on existing projects is easy and relevant. Typescript was created to enhance JavaScript code [8]. Since components of Typescript can be reused it reduces the problem of code duplication. Typescript can be used on both the frontend and backend of web applications. It has become a common requirement for JavaScript developers to know typescript because many of the libraries and frameworks of JavaScript adopt Typescript by default. Large codebase, need for speed and team used to with static typing are the typical use cases of TypeScript. A survey conducted among 65000 stack overflow developers, it is concluded that Typescript is the second most loved programming language [9].

Figure 5: Survey by stack overflow, in 2020 TypeScript has surged in popularity - source: [9]
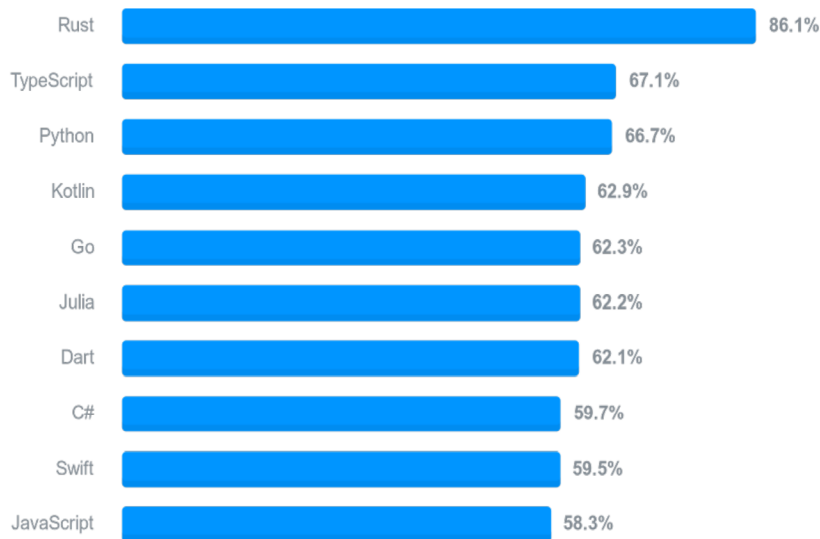
As shown in figure 2, TypeScript is second most loved programming language leaving python on third.

3.5.1   Type System of TypeScript

There are a number of sophisticated constructions and notions in the typescript type system. These include types for object-based programming, structural type equivalence, gradual typing, subtyping of recursive types, and type operators [18]. These features taken together should make for a very pleasant programming experience. One of the type system's main design goals is to support current JavaScript styles and idioms, as well as to be compatible with the great majority of existing and extremely popular JavaScript modules.

Following are the distinctive properties of type System

- **Structural Types:** Rather than being nominal, the TypeScript type system is structural. TypeScript compares types using a structural type system, which means it only considers the type's members when comparing them [19]. For JavaScript programming, where objects are frequently generated from scratch and used only on the basis of their expected shape, structural type is the best fit.

- **Unified Object Types:** Objects, functions, constructors, and arrays are not distinct types of values in JavaScript; a single object can perform many functions at the same time. As a result, TypeScript object types can express not just members but

30

also call constructor and indexing signatures, which describe how the objects can be used in various ways [20].

- **Full Erasure:** In the JavaScript produced by the compiler, the types of a TypeScript program leave no trace. There are no type representations at run –time, and thus no type checking at run time. Current dynamic techniques for "type checking" in JavaScript scripts, such as checking for the presence of specific properties or the values of specific strings, aren't ideal but they're fair enough [21].

- **Type Interference:** Type interference is used by TypeScript to reduce the number of type annotations that programmers must specify directly. The logic of JavaScript should not be clouded by unnecessary new syntax because it is a terse language. In reality, only a few type annotations are usually required for the compiler to infer meaningful type signatures [22].

- **Gradual Typing:** TypeScript is an example of a gradual type system, in which certain parts of a program are statistically typed and others are dynamically typed using a distinct dynamic type, written as "*any*". Gradual typing is normally implemented via run-time but due to type erasure, that is not possible in TypeScript. As a result, type errors that aren't discovered statistically may go undetected during run-time [23].

3.5.2   TypeScript Annotation and Inferences

Two TypeScript systems for referring value types to variables are Annotation and Inference. Both of these inform you of the value type that should be assigned to a variable. The key distinction is that you must manually assign annotations to the annotations, whereas the Inference will do so automatically.

Source code 7: Example of annotation

```
var age: number = 20;
var  name : string = 20;
```

When the source code 7 gets executed, the first line of code declares variable as age and in the next line variable name is declared as string but the assigned value is number. The result of the variable name is '20'. When compiling the preceding program, the TypeScript compiler correctly detects and reports the type mismatch.

The TypeScript inference system automates the process of determining and assigning a type to a variable. All of the examples in the annotation section above could be written without the annotations, and all of the types would still be allocated correctly.

Source code 8: Typescript inference system

```
const age = 25;
const name = 'Harry';
```

Source code 8 defines interference system of typescript. TypeScript is intelligent enough to automatically evaluate and assign the best possible matching types. Except in circumstances when unknown return values from functions or variables must be specified before initialization, you don't need to bother about manually creating annotations most of the time [38].

3.5.3   Primitives

Simple types are also known as primitive types, and they are TypeScript's built-in predefined types. "Number", "string", "boolean", "null type", "undefined type" and "any type" are the primitive data types. The null primitive represents a null literal and it is not possible to directly reference the null type value itself. The undefined type is the type of the undefined literal. All kinds are sub-types of the undefined type. Any is a special type that can be used to represent any value. Any type is a subtype of any other type, and any can be assigned to and from any other type. If a variable is specified as having the type any, type verification for that variable is effectively disabled [39].

Source code 9: object with type any

```
let obj: any = {x: 0};
// None of the following lines of code will throw compiler errors.
// Using `any` disables all further type checking, and it is assumed
// you know the environment better than TypeScript.
obj.name();
obj();
obj.age = 20;
obj = 'Harry';
const n: number = obj;
```

After the execution of source code 9, the typescript compiler won't throw a type error because of type "any".

3.5.4 TypeScript Objects

Any value that isn't a primitive value is represented by the TypeScript object type. The Object type, on the other hand, describes the functionality that is common to all objects. An object of the empty type {} is one that has no properties of its own.

Source code 10: object with predefined types

```typescript
const employee: {
  firstName: string;
  lastName: string;
  age: number;
} = {
  firstName: 'Harry',
  lastName: 'Sen',
  age: 25,
};
```

When the above source code 10 gets executed, it defines employee object with types. In the above example, the employee object is an object type with a fixed list of properties. If we try to access property that does not exist on an employee object then it throws an error.

3.5.5 TypeScript Functions

The program's functions ensure that it is maintained and reusable, as well as arranged into understandable blocks. Despite the fact that TypeScript has the concept of classes and modules, functions are still an important aspect of the language. There are two sorts of functions in TypeScript: named and anonymous.

Source code 11: Typescript functions

```typescript
function Sum(x: number, y: number): number {
  return x + y;
}
Sum(5, 4); // returns 9
```

When the source code 11 gets executed, it will have type annotations for both argument type and return type. All of the type annotations, like variable declarations, can be omitted. The TypeScript compiler can generally deduce the return type if the return type declaration is omitted. The TypeScript compiler will assign the type any if the type annotations for the argument types are removed.

3.5.6 Arrow Functions

For anonymous functions, or function expressions, fat arrow notations are employed. We eliminated the necessity for the function keyword by using fat arrow =>. The function expression is enclosed between the curly brackets {}, and the parameters are passed in parentheses ().

Source code 12: Arrow function in typescript

```
const sum = (x: number, y: number): number => x + y;
sum(5, 4); // returns 9
```

The above arrow function sum will be converted to javaScript code as follows:

Source code 13: Conversion of arrow function to javascript

```
var sum = function (x, y) {
  return x + y;
 };
```

3.5.7 Function Overloading

The concept of function overloading is available in TypeScript. Multiple functions with the same name but different parameter types and return types are possible. The number of parameters, however, should be the same.

Source code 14: function overloading

```
function add(a: string, b: string): string;
function add(a: number, b: number): number;
function add(a: any, b: any): any {
 return a + b;
}
add('Hello ', 'world'); // returns "Hello world"
add(2, 3); // returns 5
```

When the above source code 14 gets executed, it will define add function. We have the same function add() with two function declarations and one function implementation in the previous example. The first signature has two string-type parameters, while the second signature has two number-type parameters. The function implementation should be in the last function. Because the return type in the first two function declarations might be either string or number, we must utilize compatible parameters and return types in the function definition. It is not possible to overload a function with varying numbers of parameters and types with the same name.

3.5.8 TypeScript Rest Parameters

Rest parameters were added by TypeScript to make it easier to support a large number of parameters. We can use rest parameters when the number of parameters that a function will get is unknown or varies. The "arguments" variable in JavaScript is used to accomplish this. However, using TypeScript, we may use the ellipsis-denoted rest parameter '...'.

Source code 15: Rest parameters in typescript

```typescript
function Greet(greeting: string, ...names: string[]) {
  return `${greeting} ${names.join(', ')}!`;
}
Greet('Hi', 'Harry', 'Sen'); // returns "Hello Harry, Sen!"
Greet('Hi'); // returns "Hi !"
```

Source code 15 shows the usuage of rest parameters. We have a function with two parameters in the example above: greeting and names. The remaining parameter, names, is indicated by ellipses.... We first pass "Harry" and "Sen" as rest arguments while calling the function. The components of the names array are joined together to form a string array.As a result, it responds with "Hi Harry, Sen!" We don't supply any arguments as rest parameters in the second function call. The compiler accepts this, and hence the output is "Hi!". Rest arguments must be listed last in the function definition, else the TypeScript compiler will generate an error.

3.5.9 TypeScript Void Types

The void type indicates that there is no type at all. It's similar to the polar opposite of any type. When you have a function that doesn't return a value, you usually use the void type as the return type.

Source code 16: Function with type void

```typescript
function log(message): void {
  console.log(message);
}
```

Source code 16 defines function with void type.
The TypeScript subtyping rule states that in order for a function A to be a subtype of a function B, the return type of B must either be a subtype of the return type of A, or void if the return type of A is void.

3.5.10 Conditional Types

Conditional types aid in describing the relationship between input and output types. Conditional types looks like conditional expression.

Source code 17: Conditional types

```
SomeType extends OtherType ? TrueType : FalseType;
```

Source code 17 shows the usuage of conditional types.

If the type on the left of the extends can be assigned to the type on the right, you'll get the type in the first branch (the "true"); otherwise, you'll get the type in the second branch (the "false")

The infer keyword in conditional types allows us to infer from types we compare against in the true branch. Instead of obtaining it "manually" with an indexed access type, we could have inferred the element type in Flatten:

Source code 18: Inferred element type in Flatten

```
type Flatten<Type> = Type extends Array<infer Item> ? Item : Type;
```

Instead of describing how to acquire the element type of T within the true branch, we used the infer keyword to declare a new generic type variable named Item. This relieves us of the burden of figuring out how to go through and dissect the structure of the types we're interested in.

While given a union type, conditional types become distributive when acting on a generic type.

Source code 19: ToArray type

```
type ToArray<Type> = Type extends any ? Type[] : never;
type StrArrOrNumArr = ToArray<string | number>;
```

When the above source code 19 gets executed, it defines ToArray Type. When we use ToArray with a union type, the conditional type is applied to each member of that union.

3.5.11 Intersection types

We can combine numerous types into one with an intersection type. The structure of an object with an intersection type must include all of the kinds that make up the intersection types. It's indicated by a & symbol. In the object of an intersection type, all members of all types are required [40].

Source code 20: object of intersection type

```typescript
interface Animal {
  kind: string;
}
interface Person {
 firstName: string;
 lastName: string;
 age: number;
}
interface Employee {
 employeeCode: string;
}
const employee: Animal & Person & Employee = {
 kind: 'human',
 firstName: 'Harry',
 lastName: 'Sen',
 age: 25,
 employeeCode: '0101',
};
```

Source code 20 shows the usuage of object of inteersection type.

Each type is separated by a & symbol, as shown in the code above. Additionally, the employee object has all of the Animal, Person, and Employee properties. Each interface specifies a type for each property. We'll get error warnings if the structure doesn't match exactly.

3.5.12 Union types

Union types produce a new type that allows us to create objects with some or all of the properties of each of the types that made up the union type. The pipe | symbol is used to join multiples to form union types.

When the source code 21 gets executed, the employee object will have union type. The employee object in the code above is of the Animal | Person | Employee type, which indicates it can have some of the Animal, Person, or Employee interfaces' properties. They don't have to be all of them, but if they are, the type must match the ones in the interface. We can have two types with the same member name but distinct kinds using union types.

Source code 21: union types

```typescript
interface Animal {
  kind: string;
}
interface Person {
  firstName: string;
  lastName: string;
  age: number;
}
interface Employee {
  employeeCode: string;
}
const employee: Animal | Person | Employee = {
  kind: 'human',
  firstName: 'Harry',
  lastName: 'Sen',
  age: 25,
  employeeCode: '0101',
};
```

3.5.13 Non-nullable types

Non-nullable types are a feature of TypeScript that can be enabled using a compiler setting --strictNullChecks. Types can be made nullable either using unions or by making parameter as optional.

Source code 22: Non-nullable types

```typescript
function getLength(a: string | null) {
  if (a === null) {
    return 0;
  }
  return a.length;
}
```

Source code 23: type error on nullable type

```typescript
interface Employee {
  name?: string;
}
const e2: Employee = {
  name: undefined, // ok
};
const e1: Employee = {
  name: null, // Type 'null' is not assignable to type 'string | undefined'
};
```

When the source code 22 gets executed, the return type of getLength function is either string or null.

When the above source code 23 gets executed, it throws an type error on nullable type.

3.5.14 Index types

The compiler may check code that uses dynamic property names using index types. When the source coode 24 gets executed, the compiler verifies that the properties manufacturer and model belong to Car. The index type query operator, keyof T, is the first. The union of known, public property names of type T is called keyof T for any type T.

Source code 24: Index types

```
function pluck<T, K extends keyof T>(o: T, propertyNames: K[]): T[K][] {
  return propertyNames.map((n) => o[n]);
 }
 interface Car {
  manufacturer: string;
  model: string;
  year: number;
 }
 const taxi: Car = {
  manufacturer: 'Skoda',
  model: 'Iv',
  year: 2019,
 };
 // Manufacturer and model are both of type string,so we can pluck them both
into a typed string array
 const makeAndModel: string[] = pluck(taxi, ['manufacturer', 'model']);
 // If we try to pluck model and year, we get an array of a union type:
(string | number)[]
 const modelYear = pluck(taxi, ['model', 'year']);
```

3.5.15 Polymorphic this types

A polymorphic this type is a subclass of the containing class or interface. This is also called F-bounded polymorphism and this makes expressing hierarchical fluent interfaces a lot easier.

Source code 25: Polymorphic this type

```
Class BasicCalculator{
  public constructor(protected value: number = 0) {}
  public currentValue(): number {
    return this.value;
  }
  public add(operand: number): this {
    this.value += operand;
    return this;
  }
  public multiply(operand: number): this {
    this.value *= operand;
    return this;
  }
  // ... other operations go here ...
}
const v = new BasicCalculator(2).multiply(5).add(1).currentValue();
```

In the above example, the BasicCalculator class uses 'this' type; it can be extended and new classes can use old methods without any changes.

3.6   Type Declaration Files in TypeScript

A Type Declaration file is a TypeScript file which contains only the type declarations, not the business logic. These files are just intended to aid in the development process and are not intended to be included in the compilation. Type declaration in itself is a huge topic since the whole typescript ecosystem revolves around the Erased Type System (types) [17].

TypeScript declaration files are TypeScript files that solely include types and don't include any implementations. The keyword declare is used in TypeScript declaration files to declare the existence of a variable without giving it a value.

Example of typescript declaration file is as follows:

Source code 26: Typescript declaration file

```
interface ITest {
  concat(x: string, y: number): string;
}
declare let StringLib: ITest;
```

40

The above example of the declaration file declares a variable StringLib which has type of ITest. The return type of the variable StringLib is string whereas input type is string and number.

3.6.1 Interface declaration

An interface declaration specifies a type but has no effect during execution. For example, the interface below declares a type IPoint with members x and y of type number and string respectively.

Source code 27: Interface defining types

```
interface ITest {
  x: number;
  y: string;
 }
```

From the above example, any object will satisfy the ITest type if it has properties x and y with numeric and string values.

3.6.2 Class declaration

```
class Test {
  x: number;
  y: string;
  constructor (x: number, y: string) {
    this.x = x;
    this.y = y;
  }
 }
```

Above example declares a type Test and can be exposed as a variable Test during run time. So, the class Test can be instantiated by an expression new Test.

3.6.3 Ambient Class Declaration

The ambient class may be used in the case of above example. It tells the compiler to act as if the class declaration were there (for type verification purposes), but code will be generated.

Source code 28: Ambient class declaration

```
declare class Test {
  x: number;
  y: string;
  constructor(x: number, y: string);
 }
```

When the source code 28 gets executed, it declares ambient class Test.

3.7   Errors in TypeScript Declaration Files

Many different types of mistakes can arise in TypeScript declaration files. Some of them are as follows

- **Typos:** This type of error occurs when the author of the declaration file has mistyped some name.
- **Dead types**: Libraries are updated, and library declaration files must be modified to reflect the new library implementations. Authors may neglect to remove features that are no longer present in the library implementation during the process.
- **Missing types:** When constructing a TypeScript declaration file, it's common for developers to forget to document some aspects of a library.
- **Static vs. Instance Fields:** Classes in TypeScript have both static and instance fields. An author of a declaration file may forget to declare a field as static by accident.
- **Wrong types:** Occasionally, the declared types in a declaration file are incorrect. When utilizing the implementation, a variable may be specified as having the type string yet contain a number.
- **Wrong location:** It's possible that the correct type is documented in the incorrect location. Such an error can occur, for example, in TypeScript declaration files with several nested modules, if the authors were unsure where a library feature belonged.
- **Wrong inheritance:** In bigger libraries, inheritance between classes can be difficult, especially if a few classes deviate from the overall pattern of most classes extending from a common super-class. In some cases, it's possible that classes are declared to extend the incorrect super-class.

- **Syntax confusion:** Constructors are available in TypeScript for both classes and interfaces. A constructor is a JavaScript function that is called with the new keyword. The syntax for defining a constructor differs somewhat between classes and interfaces, and this variation might lead to issues if a declaration file author makes a mistake.

  Another source of syntax ambiguity is the distinction between classes and modules. In TypeScript, a class has a constructor that may be used to create an instance of the class, whereas a module does not have one. Developers sometimes make the mistake of declaring a module as a class.

- **Any type:** Sometimes variables of the declaration file may be typed as any because the author doesn't know the exact type of that variable. While implementation, that variable can be any of the types (string, boolean, number, enum) etc.

3.8   Automated testing

Automated testing is a type of dynamic analysis to test and compare the actual outcome with the expected outcome [29]. That input could be a program's specification or manually developed test cases. After the inputs have been computed, the automated tester will run the program normally, with no performance penalty compared to running the program outside of the analysis. These automated testing techniques are used to identify errors in which a software fails to meet some implicit or explicit requirements.

Source code 29: Typescript function for simple division operation

```
function safeDivision(x: number, y: number) {
  return x / y;
 }
```

An automated tester can verify that the function meets the requirements by generating a random numeric value that can be used to run the safeDivision function. For example, the automated tester might call safeDivision(72, 8) which results in the number 9. For a set number of iterations, the automated tester will test the safeDivision function with various random inputs. During these iterations, the automated tester may or may not detect an input that causes a division by zero, causing the specification to be broken. For example,

safeDivision (12, 0) will result in the JavaScript value Infinity, and the value of infinity in JavaScript is NaN.

3.9   TypeScript Over JavaScript

When creating a modern web or JavaScript-based application, TypeScript is an better choice. TypeScript's thoughtful language features and capabilities, as well as its ever-improving tools, make for a fantastically productive programming experience [53,54,55].

- **Object Oriented Programming:** TypeScript is a powerful set of Object Oriented Programming (OOP) capabilities that help maintain clean and robust code, which enhances code quality and maintainability. TypeScript code is tidy and organized.

- **Interfaces, Generics, Inheritance and method Access modifiers:** Interfaces, generics, inheritance, and access modifiers are all supported by TypeScript. A good approach to specify a contract is through interfaces. Generics aid in compile-time checking, inheritance allows new objects to inherit the properties of old objects, and access modifiers govern the members of a class's accessibility. The access modifiers public and private are available in TypeScript. The members are public by default, it can be made private or public by adding a public or private qualifier to them.

- **Strongly/Static typed:** TypeScript does not enable values of multiple datatypes to be mixed together. Errors are thrown when these constraints are broken. As a result, when declaring variables, types must be declared, and it is not possible to assign values other than the type defined, which is extremely possible in JavaScript.

- **Compile-Time/ Static Checking:** Compile-time errors are thrown by the compiler if the right syntax and semantics of any programming language are not followed. It won't be allowed to run a single line of your software unless you fix all the syntax mistakes or debug the compile-time errors. This is also the case with TypeScript.

- **Less Code compared to JavaScript:** Because TypeScript is a wrapper for JavaScript, it provides utility classes that simplify the code. Typescript code is more comprehensible.

- **Readability:** Code readability is provided through interfaces, classes, and other elements. The code is more understandable and easy to read and understand because it is expressed in classes and interfaces.

44

- **Compatibility:** Underscore.js, Lodash, and other JavaScript libraries are compatible with Typescript. They have a lot of built-in and simple-to-use features that speed up development.

- **Provide a compiler that can convert code to JavaScript equivalent code:** TypeScript code comprises normal JavaScript code as well as TypeScript-specific keywords and constructions. The TypeScript code, on the other hand, gets transformed to normal JavaScript when it is compiled. As a result, the JavaScript generated can be utilized in any browser that supports JavaScript.

- **Support Modules:** As your TypeScript codebase expands, it's necessary to arrange classes and interfaces to make it easier to manage. It can be accomplished just by using TypeScript modules. A module is a container of code that aids in the clean organization of code. In terms of concept, they're similar to.NET namespaces.

- **ES6 Feature Support:** Because Typescript is a superset of ES6, it has all of the capabilities of ES6 plus a few extras, such as the ability to use Arrow Functions, also known as lambdas. The fat arrow syntax, which was introduced in ES6, is a somewhat different way of defining anonymous functions.

- **Used in Popular Frameworks:** In the last few years, TypeScript has become increasingly popular. The time that Angular 2 officially transitioned to TypeScript, which was a win-win situation, was maybe the defining point of TypeScript's popularity.

- **Reduce Bugs:** Null handling, undefined, and other issues are reduced. Developers are unable to write type-specific code with sufficient checks due to strongly typed characteristics.

- **Function Overloading:** Overloaded functions are possible in TypeScript. This manner, depending on the parameter, you can call several implementations of a function. However, keep in mind that TypeScript function overloading is a little strange and necessitates type verification during implementation. This limitation arises from the fact that TypeScript code is eventually compiled into normal JavaScript, which does not enable function overloading in the proper sense.

- **Constructors:** In TypeScript, you can declare constructors for the classes you create. The constructor is typically used to initialize an object by assigning default values to its properties. The constructor, like a function, can be overloaded.

- **Debugging:** The code built in TypeScript is simple to debug.

- **TypeScript is just JavaScript:** JavaScript is the starting point for TypeScript, and JavaScript is the ending point for TypeScript. The core building parts of your application are taken from JavaScript by Typescript. For the purpose of execution, all TypeScript code is translated to its JavaScript equivalent.

- **Portable:** TypeScript is cross-browser, cross-device, and cross-OS compatible. It will work in any JavaScript-enabled environment. TypeScript does not require a dedicated VM or a specific runtime environment to run, unlike its competitors.

3.10 Related works

3.10.1 TSCheck

TSCheck is a tool for finding bugs in hand-written TypeScript type definitions, which works by comparing the type definitions to the actual JavaScript library implementation [24]. TSCheck work in three phases: It runs the library's initialization code and takes a snapshot of the runtime state that results; it then compares the TypeScript type declarations to the objects in the snapshot, which describe the library API'S structure; finally, each library function undergoes a light-weight static analysis to type check the return value of each function signature. Experimental results demonstrated on [25] show 142 errors in the declaration files of 10 libraries, with an analysis time of a few minutes per library and with a low number of false positives. Additionally, the way developers use library interface declarations indicates several TypeScript type system constraints. In contrast to TSCheck our typechecker script finds type error that is ignored by Javascript library's type definition and API structure.

3.10.2 TypeScript TPD

TypeScript TPD is a tool for use with TypeScript, based on the polymorphic blame calculus, for monitoring JavaScript libraries and TypeScript clients against the TypeScript definition [29]. TPD was not created with the intent of detecting erroneous definition files but using the DefinitelyTyped method of assessed progressive typing, we may detect definition problems. TPD is a progressive typing application that covers JavaScript libraries based on their TypeScript definition files. In TPD wrappers using proxies were implemented to attach type checking code without rewriting to the library. TPD was able

to detect a significant number of mismatches between libraries and their definition files. In contrast to TPD our typechecker compares the types written on declaration file and type that is necessary for execution of library.

3.10.3 Optional and gradual typing

The unsoundness caused by bivariate function subtyping and method overriding does not seem to help programmers [30]. They come to the conclusion that the majority of cases of insanity are justifiable. A method for evaluating the performance of gradually-typed programming languages is introduced on [31]. If other language implementations show similar performance, the future of progressive typing could be jeopardized. They test gradual typing with Typed Racket, a language that has perhaps the most comprehensive support for gradual typing, including transparent proxies [32]. Their implementation is done for Racket programming whereas ours is done for javascript programming.

3.10.4 TSInfer and TSEvolve

TSInfer and TSEvolve are used for checking conformance between JavaScript libraries and their TypeScript definition files. Rather than performing dynamic monitoring, they conduct static analysis. They mix heap snapshot analysis with lightweight static function definition analysis. Both strategies are ineffective, but they have the advantage of requiring no run-time resources and generating no disturbance. The key idea of their tool is that many declaration errors can be detected by an analysis of the library initialization state combined with a light-weight static analysis of the library function code [33]. A run time based strategy, such as the one we offer in this paper, will provide more accurate response.

4 Practical Part

4.1 Motivation and Idea

Typescript simply verifies types during the compilation time but not during the run time. Application crashes many times because of invalid data or null data that's coming from static files, libraries or AJAX response. Developers have to write type guard manually to get rid of runtime error.

Source code 30: memories coming as server response.

```
return (
  <main className='App'>
    <h1>My Memories</h1>
    <AddMemory saveMemory={handleSaveMemory} />
    {memories.map((memory) => (
      <PostMemory
        key={memory._id}
        updateMemory={handleUpdateMemory}
        deleteMemory={handleDeleteMemory}
        memory={memory}
      />
    ))}
  </main>
)
```

When the above source code 30 gets executed, typescript doesn't throw any error during compile time but throws an error during run time.

Figure 6: Run time error because of undefined data

Figure 5 shows the runtime error caused because of undefined data. To get rid of that error using conditional checking and checking types manually can be a solution but its tedious and time consuming in the case of large applications.

Source code 31: conditional statement for memories data

```
{ memories && memories.map((post: IMemory) => (
  <MemoryItem
    key={memory._id}
    updateMemory={handleUpdateMemory}
    deleteMemory={handleDeleteMemory}
    memory={memory}
  />
))}
```

When the above source code 31 gets executed, it at first checks for the memories and renders the data accordingly.

Source code 32: Interface defining types of memory

```
interface IMemory {
  _id: string
  title: string
  message: string
  creator: string
  selectedFile: string
  status: boolean
  createdAt?: string
  updatedAt?: string
}
```

When the source code 32 gets executed, it defines Imemory with its types. So the use of automatic type checker to check the types returned by static files, libraries or api response makes the situation better.

4.2 Tools

To meet the objective various tools were used which makes the task easier, robust and flexible.

4.2.1 VsCode

Visual studio code is a lightweight but powerful source code editor which run on your desktop and is available for windows, macOs and Linux [41]. It contains built-in JavaScript, Typescript and node.js support, as well as a large community of extensions of other languages. Support for debugging, syntax highlighting,

intelligent code completion, snippets, code refactoring, and embedded Git are just a few of the features supported by VsCode. According to the survey done by stack overflow [42], the most common developer environment tool is visual studio code, which is used by 70 % of the 82,000 respondents.

4.2.2 Faker.js

Faker.js is a popular javascript library that generates fictitious (but plausible) data that may be used for variety of puroposes like Unit testing, Performance testing, Building demos, working without backend. The data generated by faker can be used on both server-side as well as browser. The faker.js has API support for various areas, including address, animal, commerce, finance, image, localization [47].

4.2.3 Recharts

Recharts is a popular data visualization and charting packages for ReactJs. Recharts was one of the first charting libraries developed just for use in ReactJs apps, with the library itself being constructed using react and D3js under the hood. It was first released in 2016 and one of the first charting libraries designed specifically for use in ReactJs apps. The main principles of Recharts are: simply deploy with react components; native SVG support, lightweight depending only on some D3 submodules; declarative components, components of charts are purely presentational [48].

4.2.4 Material UI

Material UI abbreviated as MUI is a react UI library. MUI is a comprehensive, flexible, and easily available library of foundation and advanced components that allows us to create our own design systems and construct React apps more quickly [43]. Material-UI components do not pollute the global scope and work without any additional configuration.

4.2.5 MongoDB

MongoDB is a document-oriented database that is open source and cross-platform. MongoDB is a NoSQL database application that works with JSON like documents and optional schemas. It is a database created by MongoDB Inc. and is distributed under the server side public license [44]. MongoDB cloud provides different clusters for storing data.

4.2.6 Express

Express is a node.js web application framework that offers a comprehensive range of functionality for both web and mobile apps. Using a variety of HTTP utility methods and middleware, we can quickly and easily build a powerful API. Express adds a thin layer of web application functionality without obscuring the node.js capabilities that we are already familiar with [45].

4.2.7 Axios

For node.js and browser Axios is a promise based HTTP client. Axios is a short package with a very extendable interface that gives a simple to use library in a compact package. It is isomorphic meaning, with the same codebase it can run in browser and node.js. It uses the native node.js http module on the server, and XMLHttpRequest on the client (browser) [46].

4.3 Experimental application

This Experimental application used in practical part entitled SMRS (Sweet memories recording system) is aimed to use type checker script during run time.

The script will contribute to make sure any objects received from a library, static file or from any AJAX response matched the expected type of object. However, it also prohibits developers from providing invalid arguments.

The application is able to store sweet memories. The layout and structure of application looks same like the social media memories. Using the application user can store their memories with description and photos and update or delete them. The experimental

application also renders the chart which is used for the evaluation of our type checker script.

Figure 7: SMRS user interface
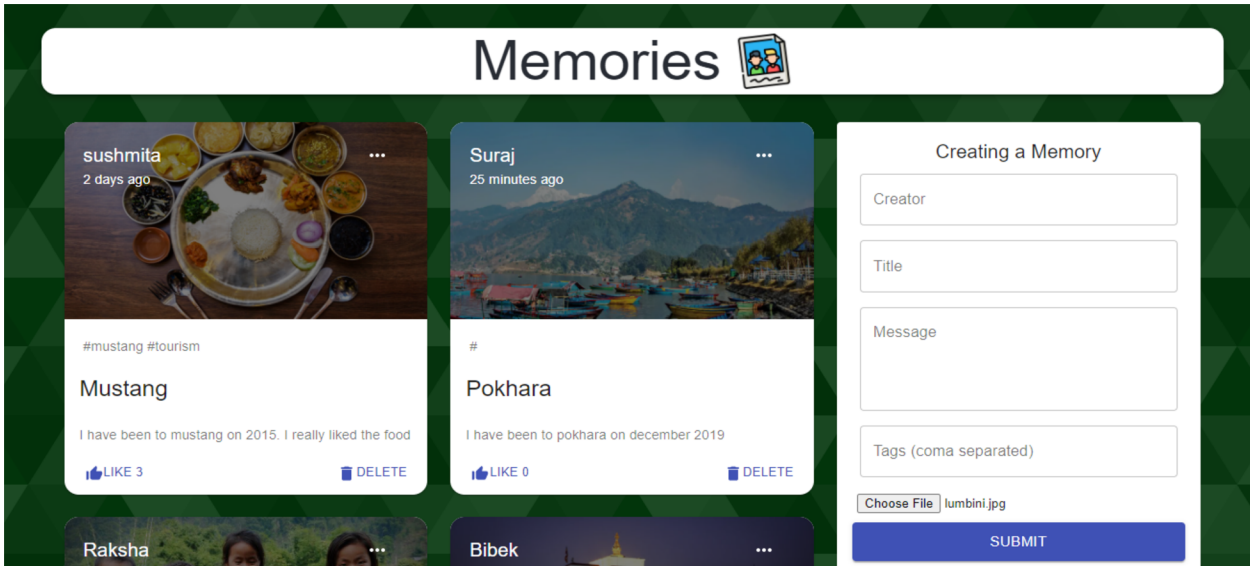
Figure 6 shows the user interface of developed sweet memories recording system.

On SMRS there are in total three components namely, formComponent, memoryComponent, rechartComponent.

4.3.1 FormComponent

FormComponent is used to provide a platform to user from where user can upload their memories detail on database. The role of this component is to render form and capture the state of data.

Source code 33: Form component as functional component

```
const AddMemory: React.FC<Props> = ({ saveMemory }) => {
  const [formData, setFormData] = useState<IMemory | {}>()

  const handleForm = (e: React.FormEvent<HTMLInputElement>): void => {
    setFormData({
      ...formData,
      [e.currentTarget.id]: e.currentTarget.value,
    })
  }
}
```

In the above code, we can see FormComponent is functional component of React which only accepts children of type Props. State of form data are captured by react-hooks. Every onchange event of form elements are captured by handleForm function. After user submits the form saveMemory function gets called.

Source code 34: SaveMemory function

```
const handleSaveMemory= (e: React.FormEvent, formData: IMemory): void => {
  e.preventDefault()
  addMemory(formData)
  .then(({ status, data }) => {
   if (status !== 201) {
     throw new Error('Error! Memory not saved')
   }
   setPosts(data.posts)
 })
 .catch((err) => console.log(err))
}
```

HandleSaveMemory only accepts the data of type IMemory. If it doesn't match the type then it returns type error on compile time. For connection with server submit function call addMemory Api for exchanging data with Server.

Source code 35: Add memory function of API

```
export const addMemory = async (
  formData: IMemory
): Promise<AxiosResponse<ApiDataType>> => {
  try {
    const memory: Omit<IMemory, '_id'> = {
      title: formData.title,
      message: formData.message,
      creator: formData.creator,
      selectedFile: formData.selectedFile,
      status: false,
    }
    const savePost: AxiosResponse<ApiDataType> = await axios.post(
      baseUrl +'/memories,
      memory
    )
    return saveMemory
  } catch (error) {
    throw new Error('error')
  }
}
```

In the above code, all the data of form are sent to server as POST request and server verifies the data. Axios package of npm has been used as promise based HTTP client for browser and node.js. Axios also provides client side support for cross-site request forgery.

Source code 36: Create Memory function in server

```
export const createMemory = async (req: Request, res:Response) => {

  const { title, message, selectedFile, creator, tags } = req.body;
  const newMemories = new Memories({ title, message, selectedFile, creator,
tags })

  try {
      await newMemories.save();
      res.status(201).json(newMemories );
  } catch (error) {
      res.status(409)
  }
}
```

In the above code, once the api calls post route, express of node.js verifies the request data. Once the data are verified it saves the data on mongodb database cluster using mongoose. Once the operation is done either success/error of the response status and data are sent back to browser.

Source code 37: Update memory function in API

```
export const updateMemory = async (
  memory: IMemory
): Promise<AxiosResponse<ApiDataType>> => {
  try {
    const memoryUpdate: Pick<IMemory, 'status'> = {
      status: true,
    }
    const updatedMemory: AxiosResponse<ApiDataType> = await axios.put(
      `${baseUrl}/memories/${memory._id}`,
      memoryUpdate
    )
    return updatedMemory
  } catch (error) {
    throw new Error('error')
  }
}
```

In the above code, to update memory all the updated information from client are sent to server as PUT request and server verifies the data. Id of particular data are passed as request parameters.

Source code 38: Update memory function in Server

```
export const updateMemory = async (req: Request, res:Response) => {
  const { id } = req.params;
  const { title, message, creator, selectedFile, tags } = req.body;

  if (!mongoose.Types.ObjectId.isValid(id)) return res.status(404).send(`No Memory with id: ${id}`);

  const updatedMemory = { creator, title, message, tags, selectedFile, _id: id };

  await PostMemory.findByIdAndUpdate(id, updatedMemory, { new: true });

  res.json(updatedMemory);
}
```

In the above source code, once the api calls put route, express of node.js verifies the request data. Once the data are verified it finds the data with particular id and saves updated data on mongodb database cluster using mongoose. Once the operation is done either success/error of the response status and data are sent back to browser.

Source code 39: deleteteMemory function in API

```
export const deleteMemory = async (
  _id: string
): Promise<AxiosResponse<ApiDataType>> => {
  try {
    const deletedMemory: AxiosResponse<ApiDataType> = await axios.delete(
      `${baseUrl}/memories/${_id}`
    )
    return deletedMemory
  } catch (error) {
    throw new Error('error')
  }
}
```

In the above code, to delete memory all the information that needs to be deleted from client are sent to server as Delete request and server verifies the data. Id of particular data are passed as request parameters.

Source code 40: delete memory function in server

```
export const deleteMemory = async (req:Request, res:Response) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id)) return res.status(404).send(`No
post with id: ${id}`);

  await PostMemory.findByIdAndRemove(id);

  res.json({ message: "Memory deleted successfully." });
}
```

In the above code, once the api calls delete route, express of node.js verifies the request data. Once the data are verified it finds the data with particular id and deletes data on mongodb database cluster using mongoose. Once the operation is done either success/error of the response status and data are sent back to browser.

Source code 41: Like Memory function in the server

```
export const likeMemory = async (req:Request, res:Response) => {
  const { id } = req.params;

  if (!mongoose.Types.ObjectId.isValid(id)) return res.status(404).send(`No
memory with id: ${id}`);

  const memory = await PostMemory.findById(id);

  const updatedMemory = await PostMemory.findByIdAndUpdate(id, { likeCount:
memory.likeCount + 1 }, { new: true });

  res.json(updatedMemory);
}
```

In the above code, once the api calls likeMemory route, express of node.js verifies the request data. Once the data are verified it finds the data with particular id and updates like count on mongodb database cluster using mongoose. Once the operation is done either success/error of the response status and data are sent back to browser.

4.3.2   Memory Component

The memory component is responsible for rendering list of moments in UI. When the component mounts all the memories are fetched from server using fetch endpoint.

Source code 42: FetchMemory function in client

```
const fetchMemories = (): void => {
  getMemories()
  .then(({data}:IMemory[]| any)=>{
    setMemories(data);
  })
  .catch((err: Error) => console.log(err))
}
```

In the above source code, when the component is rendered fetchMemories calls getMemories endpoint. If data are fetched properly then it renders the data in UI otherwise it throws an error.

MemoryComponent also calls fetchMemory function to get detail information about particular memory. It then calls getMemory endpoint by passing id as its params.

Source code 43: Get Memories function in API

```
export const getMemories = async (): Promise<AxiosResponse<ApiDataType>> =>
{
  try {
    const memories: AxiosResponse<ApiDataType> = await axios.get(
      baseUrl + '/memories'
    )
    return memories
  }
 catch (error) {
    throw new Error('error')
  }
}
```

In the above code, getMemories endpoint calls the memories route of server using GET method of request. Once it gets response back from the server then it returns back memories as response to browser.

Source code 44: Get memories function in the server

```
export const getMemories = async (req: Request, res:Response) => {
  try {
      const memoryMessages = await MemoryMessage.find();

      res.status(200).json(memoryMessages);
  } catch (error) {
      res.status(404);
  }
}
```

In the above code, list of memories is searched in database cluster. Once it finds the list , it's information are stored on constant memoryMessages. Then getMemories sends back the response status and response data to browser.

Source code 45: GetMemory function in server

```
export const getMemory = async (req: Request, res: Response) => {
  const { id } = req.params;

  try {
      const memory = await MemoryMessage.findById(id);

      res.status(200).json(memory);
  } catch (error) {
      res.status(404);
  }
}
```

In the above code, memory with that particular request id is searched in database cluster. If memory with particular id exists, then it will send response status and data to browser otherwise it sends error response to browser.

4.3.3   RechartComponent

Rechart Component is used in our application to display Barchart using data generated by faker.js. The main purpose of this component is to check and verify our typechecker script available within our application. On this component we have used two different javascript libraries. Faker.js to generated random fake data and recharts.js to generate barchart.

In the source code 46, we have at first verified the types of faker data using shouldBe function. Once shouldBe function verifies it without any error we have set the data generated by faker.random function on data variable. We have also verified the types of

chartElements using checkType function. We should not render the chart if we have type mismatch. Thus, we have written condition using ternary operation. The condition says if all the types and values matches the types of chartElements then render bar chart otherwise render the error message. We could handle the error differently on console but it's out of scope on our current requirement because we are only focused on checking types and returning type error if any.

Source code 46: Rendering chart with random data

```
const fakerTypes = shouldBe <Record<string, number>> (faker.random);
  const data = [ fakerTypes &&
    { name: "Asia", likes: faker.random.number(2500), dislikes:
faker.random.number(1500) },
    { name: "Europe", likes: faker.random.number(2500), dislikes:
faker.random.number(2500) },
    { name: "America", likes: faker.random.number(2500), dislikes:
faker.random.number(2500) },
    { name: "Australia", likes: faker.random.number(2500), dislikes:
faker.random.number(2500), },
    { name: "Africa", likes: faker.random.number(2500), dislikes:
faker.random.number(2500)},
  ];
  const chartElements = checkType(OriginalBarchartType, CustomBarChartType);

{ chartElements ? (
        <BarChart
      width={600}
      height={300}
      data={data}
      margin={{ top: 5, right: 30, left: 20, bottom: 5 }}
    >
      <CartesianGrid strokeDasharray="3 3" />
      <XAxis dataKey="name" />
      <YAxis />
      <Legend />
      <Bar dataKey="likes" fill="#8884d8" />
      <Bar dataKey="dislikes" fill="#82ca9d" />
      <div> This is additional element </div>
    </BarChart>

    ):(
    <div style={{color: "red"}}>All the elements didn't match type
defined by recharts</div>
    ) }
```
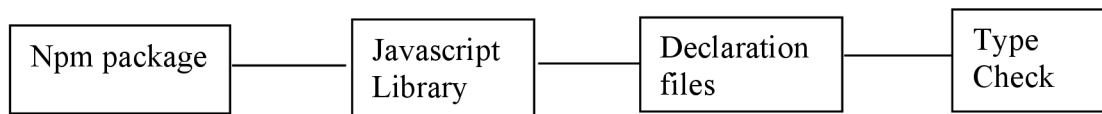
4.4 Type Checker

The concept of TypeChecker is based on run-time information that is given by javascipt libraries. TypeChecker is responsible for creating and checking the types of data that is required by particular javascript library.

Figure 8: Type checker – Architecture Overview

```
┌──────────────┐      ┌──────────────┐      ┌──────────────┐      ┌──────────────┐
│ Npm package  │──────│ Javascript   │──────│ Declaration  │──────│ Type         │
│              │      │ Library      │      │ files        │      │ Check        │
└──────────────┘      └──────────────┘      └──────────────┘      └──────────────┘
```

Above figure 7, shows the architectural overview of Type checker. As type checker is based on run-time information, we need some javascript library that is retrieved from Npm package.

4.4.1 Javascript Library

To check the type related problem in particular javascript library, we at first need to import that library in our codebase. All the dependencies and example code is provided by the developer of that library in their readme file. With the help of that readme file we can find the proper way of implemeting the library.

4.4.2 Type declaration file

Most of the latest versions of javascript libraries provide type declaration file. Some of libraries may have type 'any' which means it can ignore the type check. Typescript verifies required types of libraries from its declaration file. For our typechecking script we have generated type interface on the basis of its declaration file.

4.4.3 Type check

TypeScript doesn't have runtime type guard support for any javascript libraries. We have to write type guard manually for libraries to prevent it from crash on run time. To get rid of

this problem we have generated Type check script which checks for the type and generates type guard for it.

At first we have created a type declaration file which can be considered as type guard factory, where we have defined a function which verifies the type of value.

Source code 47: type declaration file for type check

```
/* the function verifies the type of value */
* Example cases
* if(shouldBe<string[]>)(value)){
  *   value should be array of string
*    }
if(shouldBe<Record<string, boolean>)(value){
  * value should be record of string and boolean
*}
*/

declare function shouldBe<T>(value: unknown): value is T;

export {shouldBe}
```

In the above source code the function shouldBe verifies the type of value. It returns true if the value is assigned to type T otherwise false and will result type error. The example usuagae of shouldBe function is described briefly on the documentation of source code. From the above type guary factory we can generate individual guard. In real application we can have collections of data with same data types. Thus we can use the generated guard repetedly with simple import statement.

Source code 48: TypeCheck implementation to verify type and values

```typescript
import {customCompareEquality, doArrayMatch, doObjectMatch, isAObject}
from "./shared";

export const shouldBe = <T>(value: T): value is T =>{
  if (value && Object.keys(value).length >0 ) {
      return true;
  }
  return false;
};
export type EqualityType = (
  objectA: any,
  objectB: any,
) => boolean

export type CompareEqualityTypes = (value: EqualityType ) =>
customCompareEquality;

function performCheckType(checkIsEqual?:
CompareEqualityTypes):EqualityType{
    const isEqual : customCompareEquality = typeof checkIsEqual ===
'function'?
    checkIsEqual(doComparision):(a: any, b: any) => doComparision(a, b);

    function doComparision (a:any, b:any){
      if(a === b ){
        return true;
      }
      if ( a && b && typeof a === 'object' && typeof  b == "object"){
        if (isAObject (a) && isAObject(b)){
        return  doObjectMatch(a, b, isEqual);
        }

      let aType = Array.isArray(a);
      let bType = Array.isArray(b);

      if(aType || bType){
        return aType === bType && doArrayMatch(a, b, isEqual);
      }
      return doObjectMatch(a,b, isEqual);
    }
    return a !== a && b!== b;
    }
    return doComparision

}
export const checkType  = performCheckType();
```

62

In the above source code 48, we have checkType constant which executes performCheckType function and shouldBe function returns the type predicate.

To make PerformCheckType function workable we have to pass two parameters. The two parameters are then compared using doComparision function. doComparision function at first verifies the type of both parameters. In our case, it assumes either the parameters will have type string or object or object of array. If both of the parameters don't have same types it will return false which means there is type mismatch. On the other hand doComparision function also compares the the values of both parameters and types of nested object.

Thus, checkType does comparision between provided types by javascript libraries and types that is required for execution of particular library and returns boolean true or false.

4.4.3.1 Type check on Rechart Library

Recharts verifies the type of the element being rendered on DOM and ignores if it doesn't match the required types. It doesn't show any error or warning in the console. This is good in some respects because unintentionally rendering the wrong component does not disrupt the UI, but it is not the best developer experience because we don't get any warnings on the console when we try to render an improper component, even in developer mode.

Source code 49: Barchart of Recharts library

```
 <BarChart
width={600}
height={300}
data={data}
margin={{ top: 5, right: 30, left: 20, bottom: 5 }}
>
<CartesianGrid strokeDasharray="3 3" />
<XAxis dataKey="name" />
<YAxis />
<Legend />
<Bar dataKey="likes" fill="#8884d8" />
<Bar dataKey="dislikes" fill="#82ca9d" />
<div> This is additional element </div>
</BarChart>
```

In the above source code, we have additional div element inside BarChart component. It shows no error either on run time or on compile time. While rendering the component it

completely ignores the div element and renders the elements that are defined on rechart types.

Figure 9: Chart rendered by source code 48
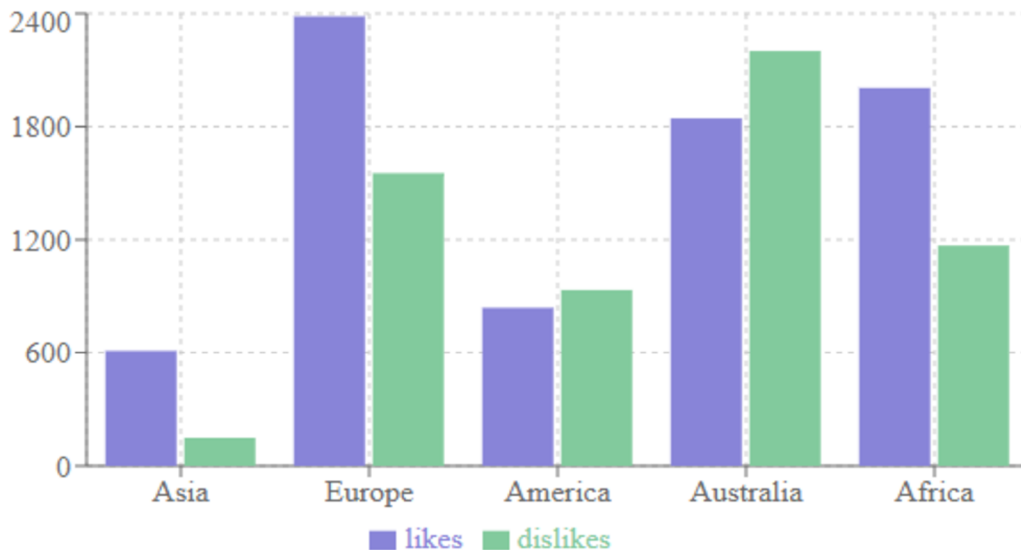
In the above figure, we can't see div rendered inside the chart component. This is more tricky if we want to render custom elements on chart component because of chart component not throwing any error.

To get rid of the above mentioned problem we can use our generated typechecker as shown in sorce code 50.

In the below source code 50, we have at first done typecheck for the elements of chart and if it satisfies all the type check then only it renders chart component otherwise renders error message.

Figure 10: Error message rendered by source code 50

All the elements didn't match type defined by recharts

Figure 10 shows the error message generated by source code. Source coude 48 generated error message because we have extra div element inside chart component.

Source code 50: Use of type checker on rechart components

```
const chartElements = checkType(OriginalBarchart, CustomBarChart);
 return (
     <>
{ chartElements ? (
           <BarChart
         width={600}
         height={300}
         data={data}
         margin={{ top: 5, right: 30, left: 20, bottom: 5 }}
      >
         <CartesianGrid strokeDasharray="3 3" />
         <XAxis dataKey="name" />
         <YAxis />
         <Legend />
         <Bar dataKey="likes" fill="#8884d8" />
         <Bar dataKey="dislikes" fill="#82ca9d" />
         <div> This is additional element </div>
      </BarChart>):
         <div style={{color: "red"}}>All the elements didn't match type
defined by recharts</div>
   }
</>
```

In the above source code 50, before rendering BarChart of recharts we have chartElements constant which stores the boolean value returned by the checkType function. The checktype function requires two arguments, the first argument originalBarChart is of type BarChart whereas customBarChart is of type BarChart and HTMLDivElement.
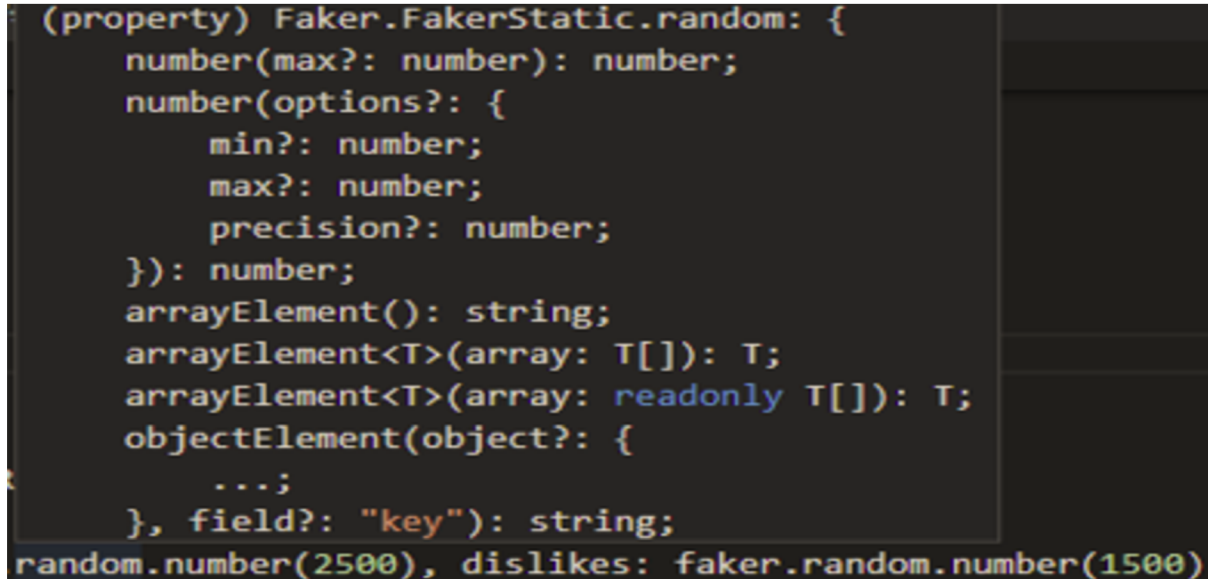
4.4.3.2 Type check for Faker.js library

Faker.js has its own declaration file for checking it's type. At first glance, it seems like types of faker library are defined properly. For the verification of it's type we have tested its types against our type checker script.

Source code 51: type check for fakers data type

```
const fakerData = shouldBe<Record<string,number>>(faker.random);
  const data = [ fakerTypes &&
    { name: "Asia", likes: faker.random.number(2500), dislikes:
faker.random.number(1500) },
    { name: "Europe", likes: faker.random.number(2500), dislikes:
faker.random.number(2500) },
    { name: "America", likes: faker.random.number(2500), dislikes:
faker.random.number(2500) },
    { name: "Australia", likes: faker.random.number(2500), dislikes:
faker.random.number(2500), },
    { name: "Africa", likes: faker.random.number(2500), dislikes:
faker.random.number(2500)},
  ];
```

In the above source code we have done typecheck for faker types. In our above case we have assumed that faker will generate record of string and number data . Our typechecker didn't return any type related errors. Which verifies that faker has correct type definition on its declaration file.

Figure 11: Type definition of faker

Above figure shows the type definition of faker for it random variable.

4.4.3.3 TypeCheck for API responses

CheckType script can be used to check types of API responses too. This script helps to verify the types of data fetched as API resposne. It's important for developers to verify type of each and every data before passing it to any library or UI.

For an example we have an interface defined with its type as follows

Source code 52: Interface Memory definining types

```
interface IMemory {
  _id: string
  title: string
  message: string
  creator: string
  selectedFile: string
  status: boolean
  createdAt: string
  updatedAt: string
}
```

When the source code 51 gets executed, interface Imemory with listed types will be defined. To check the types of value returned by API we have to do type checking as follows:

Source code 53: Function to check type manually

```
export const memories = (value: any): value is IMemory =>
  !!value &&
  typeof value.title === 'string' &&
  typeof value.message === 'string' &&
  typeof value.creator === 'string' &&
  typeof value.selectedFile === 'string' &&
  typeof value.status === 'boolean' &&
  typeof value.createdAt === 'string' &&
  typeof value.updatedAt === 'string';
```

The above code seems to be annoying, lengthy and time consuming. Not only defining guards on our own, we also have to ensure that the types and type guards do not drift apart as the code develops.

To get rid of the aforementioned problem we can use our custom developed checkType function as

Source code 54: use of typechek function

```
export const memories = shouldBe<<Record<string, boolean>>( IMemory);
```

With the use of shouldBe, we don't need to care much about type guards and we don't need to repeat ourself doing type check for individual types. We can also do type checking directly in our source code using shouldBe function

Source code 55: type check on source code

```
/* in myMemories.tsx file */
 import {shouldBe} from './typeCheck'

 if(shouldBe<memories>(data)){
     return(
       <>
       <h2> {data.title}
       <span> {data.mesasge} </span>
       </>
     )
 }
```

In the above code, we have directly tested for type guard on our source code without storing it on any variable or passing as a parameter. If we are sure that same type check won't be used more on our code base, then we can use shouldBe function.

4.5 Evaluation

After generating the script to check type of libraries, we have to evaluate the usuage of script. On the first glance, we can clearly see that script is capable of checking type of libraries / static files/ Api responses as well as it can generate type guard. But for the script to work properly, developer need to define type strictly that should be run against our script.

For the evaluation of our script, we have run experimental application with and without generated script. We have used type definitions available on used libraries as a reference because these were the only available options for comparision. As a result, we have generated a script which does comparision among two type definitions as well as create a type guard.

For each library, we have run the TypeCheck script, i.e. comparison among the types we define depending on the structure of data and type defined on the declaration file. We have run our script against two JavaScript libraries and API responses that have been used within the application. With this approach we are not completely sure about the usefulness of TypeCheck script, but it at least gives us an indication of the usefulness of script for libraries that are not maintained frequently. If there seems to be some mismatch between type definitions of libraries then TypeCheck script can be possible solution to overcome it.

On the other hand, not to limit the scope of type checker scipt we have tested our script against different Api responses.

5 Results and Discussion

5.1 Experimental Application with TypeChecker

The developed typechecker script is the result of the thesis. The thesis is mainly focused on a typechecker script that will check and validate the required types of library. We have evaluated our script on the basis of three evaluation metrics; performance, reusability, and maintainability. Two different javascript libraries and some different API responses were taken into consideration for the evaluation of the typechecker.

5.1.1 Performance

The performance test of typechecker script is done on the basis of accuracy of results rather than loading or response time. The result of typechecker on both of the libraries are shown on above sub – chapter 4.5. Without using typechecker on recharts libraries, we were unable to see any type errors because it was ignoring all the elements which doesn't statisfy it's type definition. Seeing from the perspective of users, it was quite nice because UI didn't crash because of type error. But, seeing as a developer it's time consuming as well as tricky because of not showing type related error. It would be easier for developers if it had shown type related errors on the console or on UI.

After using our type checker script, there was an error thrown on UI. The error informs developer that there is an extra element in rechart component which is not defined in the types of recharts. On the other hand, we have tested faker.js with and without our type checker script. The result was same on both cases; which means there was no type related errors/bugs in the faker.js library. Thus, from the test done for both libraries, we can evaluate that performance of type script checker depends on the type declaration files of libraries.

5.1.2 Reusability

The script is capable to check and verify types for specific data that will be used for library or for UI. The script can also generate type guard which can be used repeatedly within our application according to the requirement. We can define our custom required type guard using the type checker scipt and store it somewhere in the application. Later, as per requirement, we can re-use that custom type guards just by importing that type guard.

On the other hand, in accordance to the results shown on sub-chapter 4.5, the re-usability of the script depends on how frequently the libraries are maintained.

5.1.3   Maintainability

The documentation written inside type checker script is human readable and in simple language. The usuage of script is also written in readme file of project. Meanwhile, the source code of script has been extracted into many different simpler function which makes source code more readable. Programmers nowadys don't prefer reading whole documentation rather they prefer using IDE, to find what exactly the source code is doing. Extraction of source code makes less lines of code in main function and can be maintained easily.

 Thus, Every developer who have knowledge of typescript can understand and maintain the script easily.

5.1.4   TypeChecker over Related works

TypeChecker looks similar to those related work mentioned on sub-chapter 3.10. The result of those related works completely depends on type declaration file whereas our typechecker always compares custom type definition with type declaration files. Most of the works are based on static analysis where as the result of this thesis is based on run-time analaysis. A detail information of related work and how the result of thesis differs from them are stated on sub-chapter 3.10.

5.1.5   Summary

While doing an evaluation of our script it was found that the result of typechecker depends on the type declaration of library. Our script was able to find type error on recharts library. It was found that recharts library didn't handle types properly. To get rid of errors Recharts had ignored all the types that don't belong on its type declaration file. In the mean time our script didn't find any error on faker.js library.

On the other hand we have also tested out script against api responses. Typechecker script written inside SMRS application always checks for the valid type of API response. Type

checker script gets executed during compile time, which makes sure that application won't have bugs or errors because of invalid API response. While testing typechecking script, the script worked well and was able to store type guard on a variable or pass it as a parameter.

Similarly, the type checking script written inside it is simple and easy to understand for any developer who has some knowledge of typescript and its working mechanism. Simple documentation about the usage with examples are written within the script to make the developer familiar with it.

5.2 Further Enhancements

The TypeCheck Script written inside experimental application could be improved in a number of ways in the future. One of them is testing the developed script on different other javascript libraries available on NPM and check whether it handles or not all the type related cases. The other possibilities might be involving other developers to check the script using their complex type structure. The third possibilities could be checking whether the developed script prevents developers from passing invalid arguments. The other possibililties could be developing the script to generate type guards as a file.

One more future enhancement is to open source the developed type checking script. It makes developer task easy regardless of writing type check manually.

6 Conclusion

The main objective of this thesis was to develop and evaluate a script for type checking of JavaScript libraries to improve the development of TypeScript application. The objective has been addressed in practical part by creating TypeCheck script, which checks and verifies the types of JavaScript libraries and also generates type guard. The developed script were then run within an experimental application to support our practical part. The first partial objective was to characterize the advantages of TypeScript over JavaScript. Sub-chapter 3.3 on literature review chapter provides general information about JavaScript. The detailed sub chapter on the same sub-chapter gives detail information about the features, scope and usage of JavaScript. Sub-chapter 3.5 with expanding sub-chapters provides fundamental as well as detail information about TypeScript. The main comparision of JavaScript over TypeScript to meet the first partial objective is done on chapter 3.9 of literature review.

The second partial objective was to Build type checker script which was also a part of main objective. The source code and usage of typechecker script has been described on sub-chapter 4.4. This subchapter shows how typechecker checks for the required types of javascript libraries and throws an error if there is any mismatch between expected and required types. Similary the type guard functionality provided by typechecker script helped to solve the second partial objective.

The third partial objective was to analyze the errors returned by different used JavaScript libraries was also a part of literature review as well as practical part. To meet this partial objective experimental application was run with and without typechecker script. Then the results are compared to analysed to usefulness of our generated script.

From the experiment above, it has been seen that some of the javascript libraries are lacking exact types and are ignoring types which didn't match their types. Our typechecker can find and verify that type error. As the demand of typescript on javascript is increasing day by day most of the libraries have their own type definition which even don't need typecheck.

It has been experimentally observed that typechecker is beneficial if the libraries are not maintained frequently and declaration files are not updated. On the other hand, typechecker seems to be useful for generating type guards, which verifies the types of data.

The result of the thesis can be used by TypeScript developers who are willing to use the javascript libraries which is not maintained frequently and need to work frequently with API responses. It can be considered that working on the points mentioned on further enhancements section of previous chapter will make our typechecker more worthy and realistic.

Bibliography

1. *Wilton, P, & McPeak, J 2009, Beginning JavaScript, John Wiley & Sons, Incorporated, Hoboken. Available from: ProQuest Ebook Central. [28 July 2021].*
2. *Ecma international [online]. [Accessed 3 March 2021]. Available from: https://262.ecma-international.org/12.0/*
3. *Ecma International. 2021. [online] Available at: <https://www.ecma-international.org/publications-and-standards/standards/ > [Accessed 15 May 2021].*
4. *Babeljs.io. 2021. What is Babel? · Babel. [online] Available at: <https://babeljs.io/docs/en> [Accessed 15 May 2021].*
5. *Developer.mozilla.org. 2021. Polyfill - MDN Web Docs Glossary: Definitions of Web-related terms | MDN. [online] Available at: <https://developer.mozilla.org/en-US/docs/Glossary/Polyfill> [Accessed 1 June 2021].*
6. *Medium. 2021. 5 Reasons to Use TypeScript with React. [online] Available at: <https://blog.bitsrc.io/5-strong-reasons-to-use-typescript-with-react-bc987da5d907> [Accessed 1 June 2021].*
7. *Tutorialspoint.com. 2021. TypeScript Tutorial. [online] Available at: <https://www.tutorialspoint.com/typescript/index.htm> [Accessed 1 June 2021].*
8. *MOTTO, TODD, 2021, Introduction to TypeScript - Ultimate Courses™. Ultimatecourses.com [online]. 2021. [Accessed 3 June 2021]. Available from: https://ultimatecourses.com/blog/typescript-introduction*
9. *https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results/*
10. *POPPER, BEN, CHANDRASEKAR, PRASHANTH, 2021, The 2020 Developer Survey results are here! - Stack Overflow Blog. Stack Overflow Blog [online]. 2021. [Accessed 3 June 2021]. Available from: https://stackoverflow.blog/2020/05/27/2020-stack-overflow-developer-survey-results/*
11. *O'GRADY, STEPHEN, 2021, The RedMonk Programming Language Rankings: January 2021. tecosystems [online].2021.[Accessed 3June 2021].Available from: https://redmonk.com/sogrady/2021/03/01/language-rankings-1-21/*
12. *Stack Overflow. 2021. Stack Overflow Developer Survey 2020. [online] Available at: < https://insights.stackoverflow.com/survey/2020#most-loved-dreaded-and-wanted> [Accessed 15 August 2021].*
13. *HANENBERG, Stefan, 2010, An experiment about static and dynamic type systems. ACM SIGPLAN Notices. 2010. Vol. 45, no. 10p. 22–35. DOI 10.1145/1932682.1869462.RODRIGUES,*

14. *Elder and TERRA, Ricardo, 2018, How do developers use dynamic features? The case of Ruby. Computer Languages, Systems & Structures. 2018. Vol. 53, p. 73–89. DOI 10.1016/j.cl.2018.02.001.*

15. *DEV Community. 2021. New JavaScript Features ECMAScript 2021 (with examples). [online] Available at: <https://dev.to/brayanarrieta/new-javascript-features-ecmascript-2021-with-examples-3hfm> [Accessed 15 August 2021].*

16. *GitHub. 2021. GitHub - DefinitelyTyped/DefinitelyTyped: The repository for high quality TypeScript type definitions.. [online] Available at: <https://github.com/DefinitelyTyped/DefinitelyTyped> [Accessed 15 August 2021].*

17. *Medium. 2021. A quick introduction to "Type Declaration" files and adding type support to your JavaScript…. [online] Available at: <https://medium.com/jspoint/typescript-type-declaration-files-4b29077c43> [Accessed 15 August 2021].*

18. *BIERMAN, Gavin, ABADI, Martín and TORGERSEN, Mads, 2014, Understanding TypeScript. ECOOP 2014 – Object-Oriented Programming Lecture Notes in Computer Science. 2014. P. 257–281. DOI 10.1007/978-3-662-44202-9_11.*

19. *GitHub. 2021. GitHub - asgerf/tscheck: Finding bugs in TypeScript type definitions.. [online] Available at: <https://github.com/asgerf/tscheck> [Accessed 15 August 2021].*

20. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/play/?strictNullChecks=true&q=19> [Accessed 15 August 2021].*

21. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/docs/handbook/2/objects.html> [Accessed 15 August 2021].*

22. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-8.html> [Accessed 15 August 2021].*

23. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/docs/handbook/> [Accessed 15 August 2021].*

24. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html> [Accessed 15 August 2021].*

25. *Møller, A. and Schwartzbach, M.I., 2012. Static program analysis. Notes. Feb.*

26. *Mariano, C. L. (2017) Benchmarking JavaScript Frameworks. Masters dissertation, 2017. doi:10.21427/ D72890*

27. *Pano, A., Graziotin, D., & Abrahamsson, P. (2018). Factors and actors leading to the adoption of a JavaScript framework. Empirical Software Engineering, 23(6), 3503-3534. https://doi.org/10.1007/s10664-018-9613-x*

28. *GIZAS, Andreas, CHRISTODOULOU, Sotiris and PAPATHEODOROU, Theodore, 2012, Comparative evaluation of javascript frameworks. Proceedings of the 21st international conference companion on World Wide Web - WWW 12 Companion. 2012. DOI 10.1145/2187980.2188103.*

29. *Williams, J., Morris, J.G., Wadler, P. and Zalewski, J., 2017. Mixed messages: Measuring conformance and non-interference in TypeScript.*

30. *MEZZETTI, Gianluca, MØLLER, Anders and STROCCO, Fabio, 2016, Type unsoundness in practice: an empirical study of Dart. Proceedings of the 12th Symposium on Dynamic Languages. 2016. DOI 10.1145/2989225.2989227.*

31. *TAKIKAWA, Asumu, FELTEY, Daniel, GREENMAN, Ben, NEW, Max S., VITEK, Jan and FELLEISEN, Matthias, 2016, Is sound gradual typing dead? ACM SIGPLAN Notices. 2016. Vol. 51, no. 1p. 456–468. DOI 10.1145/2914770.2837630.*

32. *Gariano, I.O., Servetto, M. and Potanin, A., 2019. Sound Invariant Checking Using Type Modifiers and Object Capabilities. arXiv preprint arXiv:1902.10231.*

33. *FELDTHAUS, Asger and MØLLER, Anders, 2014, Checking correctness of TypeScript interfaces for JavaScript libraries. Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications. 2014. DOI 10.1145/2660193.2660215.*

34. *DEAN, Jeffrey, GROVE, David and CHAMBERS, Craig, [no date], Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. Object-Oriented Programming Lecture Notes in Computer Science. P. 77–101. DOI 10.1007/3-540-49538-x_5.*

35. *Developer.mozilla.org. 2021. Functions - JavaScript | MDN. [online] Available at: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Functions> [Accessed 15 August 2021].*

36. *Developer.mozilla.org. 2021. Inheritance and the prototype chain - JavaScript | MDN. [online] Available at: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain> [Accessed 15 August 2021].*

37. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/docs/handbook/type-inference.html> [Accessed 15 August 2021].*

38. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html> [Accessed 15 August 2021].*

39. *Typescriptlang.org. 2021. [online] Available at: <https://www.typescriptlang.org/docs/handbook/2/everyday-types.html> [Accessed 15 August 2021].*

40. *W3schools.com. 2021. JavaScript Objects. [online] Available at: <https://www.w3schools.com/js/js_objects.asp> [Accessed 15 August 2021].*

41. *Documentation for Visual Studio Code, 2021 [online] Available at: <https://code.visualstudio.com/docs> [Accessed 15 october 2021].*

42. *Stack overflow developers survey, 2021* [online] *Available at:* <*https://insights.stackoverflow.com/survey/2021#most-popular-technologies-new-collab-tools*> *[Accessed 15 october 2021].*

43. *React table component, 2021* [online] *Available at:* <*https://mui.com/*> *[Accessed 15 december 2021].*

44. *Mongostat—MongoDB manual, 2022* [online] *Available at:* <*https://www.mongodb.com/*> *[Accessed 15 January 2022].*

45. *Template engines , 2022* [online] *Available at:* <*https://expressjs.com/*> *[Accessed 15 January 2022].*

46. *Getting started | Axios Docs , 2022* [online] *Available at:* < *https://axios-http.com/docs/intro*> *[Accessed 15 January 2022].*

47. *Getting Started | Faker, 2022. Fakerjs.dev [online], Available at:* *https://fakerjs.dev/guide/*> *[Accessed 15 January 2022].*

48. *GitHub - recharts/recharts: Redefined chart library built with React and D3, 2022. GitHub [online], Available at:* <*https://github.com/recharts/recharts*>*[Accessed 15 January 2022].*

49. *Getting Started | React, 2022. ReactJs.org [online], Available at:* *https://reactjs.org/docs/getting-started.html*> *[Accessed 15 January 2022].*

50. *CHAND, SWATEE, 2022, What Is React | JavaScript Library For Building User Interfaces | Edureka. Edureka [online]. 2022. [Accessed 22 February 2022]. Available from: https://www.edureka.co/blog/what-is-react/*

51. *CHINMAYEE, DESHPANDE, 2022, What Is React [online], Available at:* <*https://www.simplilearn.com/tutorials/reactjs-tutorial/what-is-reactjs*>*[Accessed 15 January 2022].*

52. *Social Network for Programmers and Developers, 2022. Morioh.com [online], Available at:* <*https://morioh.com/p/68a5f1f5b6b9*>*[Accessed 15 January 2022]*

53. *TypeScript vs. JavaScript, 2022. Medium [online], Available at:* < *https://medium.com/geekculture/typescript-vs-javascript-e5af7ab5a331*> *[Accessed 16 February 2022]*

54. *HUMBLE, CHARLES, GALL, RICHARD and GAIN, B., 2022, How TypeScript Won Over Developers and JavaScript Frameworks - The New Stack. The New Stack [online]. 2022. [Accessed 16 March 2022]. Available from:* *https://thenewstack.io/how-typescript-won-over-developers-and-javascript-frameworks/*

55. *TypeScript vs JavaScript: Which One Should You Choose?, 2022. Radixweb [online], Available at:*< *https://radixweb.com/blog/typescript-vs-javascript*> *[Accessed 16 February 2022]*