

11. Typescript

| | |
|---------------------------------------|-----------|
| Introduction | 2 |
| Reasons of usage | 2 |
| Preventing bugs | 2 |
| Better development experience | 3 |
| Code quality | 4 |
| Easy to start and adopt | 4 |
| Community/Popularity | 4 |
| Fundamentals | 5 |
| Types | 5 |
| The “any” type | 5 |
| The “unknown” type | 5 |
| Built-in types | 6 |
| User-defined Types | 6 |
| Array | 7 |
| Tuples | 7 |
| Enums | 8 |
| Type assertions | 8 |
| Interfaces | 9 |
| Optional Properties | 9 |
| Readonly properties | 10 |
| Extending Interfaces | 10 |
| Classes | 10 |
| Inheritance | 11 |
| Functions | 11 |
| Inferring the types | 12 |
| Optional parameters | 12 |
| Default parameters | 13 |
| Unions and Intersection Types | 13 |
| Union Types | 14 |
| Intersection Types | 14 |
| Generics | 15 |
| Utility types | 15 |
| Working with Third-Party Types | 15 |
| Resources | 18 |

Introduction

TypeScript is a programming language, a strict syntactical superset of JavaScript that transpiles into plain JavaScript and adds optional static typing to the language.

The language itself is developed and maintained by Microsoft. TypeScript was first published in October 2012, after two years of internal development at Microsoft.

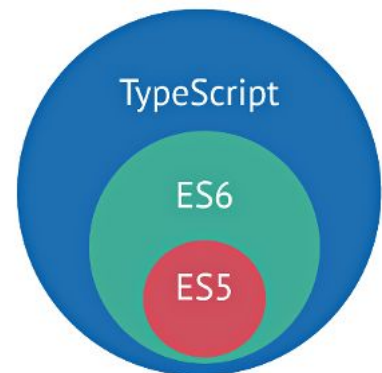


TypeScript can be used for developing JavaScript applications for both client-side and server-side applications as well.

The two most used options of transpiling TypeScript into JavaScript are default TypeScript checker (*tsc*) or Babel compiler can be invoked for the process of conversion.

TypeScript adds many features to ECMAScript 6 including:

- Type annotations and compile-time type checking
- Type inference
- Type erasure
- Interfaces
- Enumerated types
- Generics
- Namespaces
- Tuples
- Async/await



Some other features are backported from ECMAScript 5:

- Classes
- Modules
- Arrow syntax for anonymous functions
- Optional parameters and default parameters

Reasons of usage

Preventing bugs

Saying this, of course, TypeScript will not make you software bug free, **but** it can prevent a lot of type-related errors by providing static checks.

[Top 10 JavaScript errors from 1000+ projects:](#)

- Uncaught TypeError: Cannot read property
- TypeError: 'undefined' is not an object (evaluating

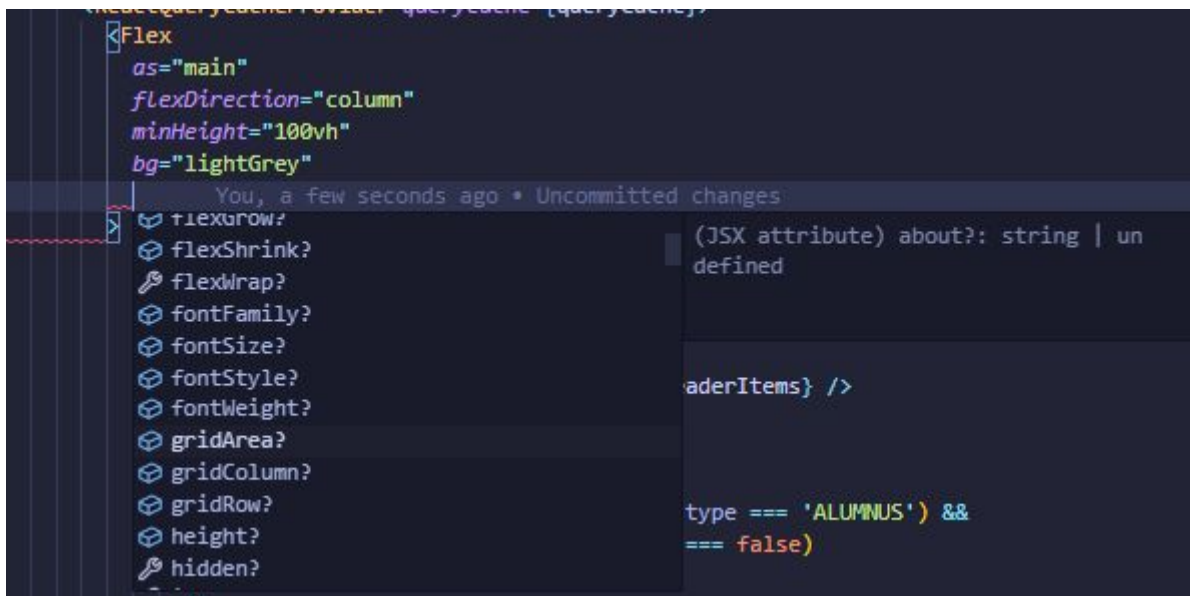
- TypeError: null is not an object (evaluating
- *(unknown): Script error*
- TypeError: Object doesn't support property
- TypeError: 'undefined' is not a function
- *Uncaught RangeError*
- TypeError: Cannot read property 'length'
- Uncaught TypeError: Cannot set property
- *ReferenceError: event is not defined*

Most of the presented errors are about mixing up your types, accessing wrong/non-existent variables or object properties, calling 'undefined' function, etc.

All these errors could be prevented by using TypeScript as it's safe typed strictness warns us about the possibility of running into an error.

Better development experience

Often, when writing an application in JavaScript, you might wonder “what arguments this function accepts?”, “what fields are in that object?” or “what properties this component accepts?”. These things can slow your development process down a lot, especially while working with new packages and libraries. You have to go through multiple files to find where the values come from sometimes. With TypeScript integration in your IDE, these problems are the past.



TypeScript can also save your time because instead of having to lookup documentations of libraries, TypeScript can suggest to you all the available options, e.g. properties in components of functions/classes.

```

store.
  // If
  if (!i
    toast
    stor
    retu
  } else {
    return decodedToken:
  }

```

(property) get: <T>(key: string, de
 faultValue?: T | undefined) => T |
 undefined

Code quality

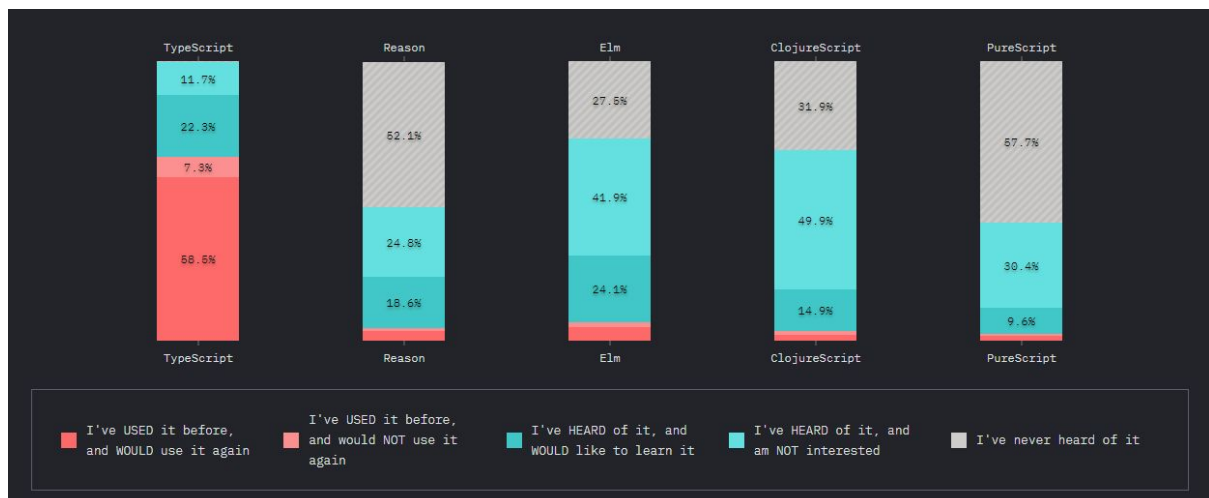
Defining data structures in the beginning, using types and interfaces, forces you to think about your app's data structure from the start and make better design decisions.

Easy to start and adopt

If you want to use TypeScript, it's very easy to get started with it. Or maybe you already are developing an application for some time that's already written in plain JavaScript? You can introduce TS to your existing project incrementally because it also compiles .js files. No need to rewrite your whole codebase at once. You can do it step by step.

Community/Popularity

TypeScript is getting more and more popular. It's used by tech companies like Google, Microsoft, Airbnb, Shopify, Asana, Adobe & Mozilla so we can assume that it reaches their expectations in terms of scalability - as they are developing large and complex applications.



Source: <https://2019.stateofjs.com/javascript-flavors/>

Fundamentals

Types

The type system represents the different types of values supported by the language. It checks the validity of the supplied values, before they are stored or manipulated by the application. This ensures that the code behaves as expected.

```
let isDone: boolean = false;           // Boolean
let age: number = 22;                  // Numbers
let color: string = "blue";           // String
let array: number[];                   // Array of numbers
let tuple: [string, number] = ["hello", 10]; // Tuples
let data: any;                          // Any
let u: undefined = undefined;          // Undefined
let n: null = null;                    // Null
let notSure: unknown = 4;              // Unknown
notSure = "maybe a string instead";

function sayHello(): void {           // Void
    console.log("Hello :)");
}
```

The “any” type

The any data type is the super type of all types in TypeScript. It denotes a dynamic type. Using the “any” type is equivalent to opting out of type checking a variable. It is not advised to use this type if possible.

```
let value: any;

value = true;           // OK
value = 42;             // OK
value = "Hello World"; // OK
value = [];             // OK
value = {};             // OK

value.foo.bar; // OK
value.trim();  // OK
```

The “unknown” type

TypeScript 3.0 introduced a new unknown type which is the type-safe counterpart of the “any” type.

The main difference between “unknown” and “any” is that “unknown” is much less permissive than “any”: we have to do some form of checking before performing most

operations on values of type “unknown”, whereas we don't have to do any checks before performing operations on values of type “any”.

```
let value: unknown;

let value1: unknown = value; // OK
let value2: any = value; // OK
let value3: boolean = value; // Error
let value4: number = value; // Error
let value5: string = value; // Error

value.foo.bar; // Error
value.trim(); // Error
value(); // Error
```

Built-in types

The following table shows all the build-in TypeScript types:

| Data type | Keyword | Description |
|-----------|-----------|--|
| Number | number | Double precision 64-bit floating point values. It can be used to represent both, integers and fractions. |
| String | string | Represents a sequence of Unicode characters |
| Boolean | boolean | Represents logical values, true and false |
| Void | void | Used on function return types to represent non-returning functions |
| Null | null | Represents an intentional absence of an object value. |
| Undefined | undefined | Denotes value given to all uninitialized variables |

The null and the undefined types are often a source of confusion. At first they may seem to do the same thing but the difference is in their meaning. Undefined is used when a value was not yet defined and has **no value** (that's why e.g. accessing non existing property of object returns undefined), while null is used to specify that this property exists (was defined) and it specifically has a **null (empty) value**.

User-defined Types

User-defined types include Enumerations (enums), classes, interfaces, arrays, and tuples.

Array

An array is a homogenous collection of values. To simplify, an array is a collection of values of the same data type. It is a user defined type.

For example an array of numbers or strings:

```
let list: number[] = [1, 2, 3];  
let list: string[] = ["one", "two", "three"];
```

Tuples

Sometimes there might be a need to store a collection of values of varied types. Arrays will not serve this purpose. TypeScript gives us a data type called tuple that helps to achieve such a purpose.

It represents a heterogeneous collection of values. In other words, tuples enable storing multiple fields of different types.

```
// Declare a tuple type  
let x: [string, number];  
// Initialize it  
x = ["hello", 10]; // OK  
// Initialize it incorrectly  
x = [10, "hello"]; // Error  
// Type 'number' is not assignable to type 'string'.  
// Type 'string' is not assignable to type 'number'.  
  
// Labeled tuple, new in TypeScript 4.0  
let y: [a: string, b: number];
```

Enums

Enums are one of the few features TypeScript has which is not a type-level extension of JavaScript. Enums allow a developer to define a set of named constants. TypeScript provides both numeric and string-based enums.

```
// Numeric enums
enum Direction {
  Up = 1, // "= 1" is optional, all other members will auto-increment
  Down,
  Left,
  Right
}

// String enums
enum Direction {
  Up = "UP",
  Down = "DOWN",
  Left = "LEFT",
  Right = "RIGHT"
}
```

Type assertions

Time to time you might run into a situation where you will know more about a value than TypeScript could figure out. This could happen when you know the type of some entity more specifically than its current type.

A type assertion is like a type cast in other languages, but it performs no special checking or restructuring of data. It has no runtime impact and is used purely by the compiler.

We can perform type assertion by the “**as**” syntax:

```
let someValue: unknown = "this is a string";
let strLength: number = (someValue as string).length;
```

Type assertions are a way to tell the compiler “trust me, I know what I’m doing.”

Interfaces

Interfaces define properties, methods, and events, which are the members of the interface. Interfaces contain only the declaration of the members. It is the responsibility of the deriving class to define the members. It often helps in providing a standard structure that the deriving classes would follow.

```
interface LabeledValue {
  label: string;
}

function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}

let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);
```

Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all.

```
interface LabeledValue {
  label: string;
  size?: number;           /* Optional Property */
}

function printLabel(labeledObj: LabeledValue) {
  console.log(labeledObj.label);
}

// OK
let myObj = { size: 10, label: "Size 10 Object" };
printLabel(myObj);

// OK, 'size' property is optional
let mySecObj = { label: "Size Optional Object" };
printLabel(mySecObj);

// Error, "Property 'label' is missing in type"
let myThirdObj = { size: 5 };
printLabel(mySecObj);
```

Readonly properties

Some properties should only be modifiable when an object is first created. You can specify this by putting “readonly” before the name of the property.

```
interface Point {
  readonly x: number;
  readonly y: number;
}

let p1: Point = { x: 10, y: 20 };
// Error, "Cannot assign to 'x' because it is a read-only property."
p1.x = 5;
```

Extending Interfaces

Interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components. This extendability also means that interfaces are much closer to how JS objects work.

```
interface Shape {
  color: string;
}

interface Square extends Shape {
  sideLength: number;
}

let square = {} as Square;
square.color = "blue";
square.sideLength = 10;

/* Extending multiple interfaces */
interface PenStroke {
  penWidth: number;
}

interface PenSquare extends Shape, PenStroke {
  sideLength: number;
}

let penSquare = {} as PenSquare;
penSquare.color = "blue";
penSquare.sideLength = 10;
penSquare.penWidth = 5.0;
```

Classes

Starting with ECMAScript 2015, also known as ECMAScript 6, JavaScript programmers can build their applications using this object-oriented class-based (OOP) approach.

TypeScript allows developers to use these techniques now, and compile them down to JavaScript that works across all major browsers and platforms, without having to wait for the next version of JavaScript.

```
class Greeter {
  greeting: string;

  constructor(message: string) {
    this.greeting = message;
  }

  greet() {
    return "Hello, " + this.greeting;
  }
}
```

If you're familiar with classes in other languages, you may have noticed in the above examples we haven't had to use the word `public` to accomplish this; for instance, C# requires that each member be explicitly labeled `public` to be visible. In TypeScript, each member is public by default. You may still mark a member `public` explicitly.

TypeScript also supports `private` and `protected` modifiers.

Inheritance

One of the most fundamental patterns in class-based programming is being able to extend existing classes to create new ones using inheritance.

```
class Animal {
  move(distanceInMeters: number = 0) {
    console.log(`Animal moved ${distanceInMeters}m.`);
  }
}
class Dog extends Animal {
  bark() {
    console.log("Woof! Woof!");
  }
}
const dog = new Dog();
dog.bark();
dog.move(10);
dog.bark();
```

Functions

Functions are basic building blocks of any JavaScript application. TypeScript functions can be created by using both named functions or anonymous functions. This allows you to choose the most appropriate approach for your application, whether you're building a list of functions in an API or a one-off function to hand off to another function.

```
// Named function
function add(x: number, y: number): number {
    return x + y;
}

// Anonymous function
let myAdd = function (x: number, y: number): number {
    return x + y;
};
```

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Inferring the types

TypeScript compiler can figure out the type even if you only have types on one side of the equation. It is called “contextual typing”, a form of type inference.

```
let myAdd: (baseValue: number, increment: number) => number =
function (x, y) {
    return x + y;
};
```

Optional parameters

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is undefined.

In TypeScript, every parameter is assumed to be required by the function. This doesn't mean that it can't be given null or undefined, but rather, when the function is called, the compiler will check that the user has provided a value for each parameter. The compiler also assumes that these parameters are the only parameters that will be passed to the function.

To provide optional parameters in TypeScript by adding a “?” to the end of parameters we want to be optional.

```
function buildName(firstName: string, lastName?: string) {
  if (lastName) return firstName + " " + lastName;
  else return firstName;
}

// ok, "lastName" is optional
let result1 = buildName("Bob");
// error, too many parameters
let result2 = buildName("Bob", "Adams", "Sr.");
// ok, all parameters provided
let result3 = buildName("Bob", "Adams");
```

Default parameters

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes undefined in its place. These are called default-initialized parameters.

```
function buildName(firstName: string, lastName = "Smith") {
  return firstName + " " + lastName;
}

// works correctly now, returns "Bob Smith"
let result1 = buildName("Bob");
// still works, returns "Bob Smith"
let result2 = buildName("Bob", undefined);
// error, too many parameters
let result3 = buildName("Bob", "Adams", "Sr.");
// works, returns "Bob Adams"
let result4 = buildName("Bob", "Adams");
```

Unions and Intersection Types

Sometimes, you run into use-cases where you want to compose or combine different existing types instead of creating them from scratch. To compose types in TypeScript, we have Intersection and Union types.

Union Types

An union type describes a value that can be one of several types. We use the singular OR operator “|” (bitwise OR in other languages) to separate each type, so **number | string | boolean** is the type of a value that can be a number, a string, or a boolean.

```
/**
 * Function takes a string and adds "padding" to the left.
 * If 'padding' is a string, then 'padding' is appended to the left
 * side.
 * If 'padding' is a number, then that number of spaces is added to
 * the left side.
 */
function padLeft(value: string, padding: string | number) {
  if (typeof padding === "number") {
    return Array(padding + 1).join(" ") + value;
  }
  if (typeof padding === "string") {
    return padding + value;
  }
  throw new Error(`Expected string or number, got '${padding}'.`);
}

padLeft("Hello world", 4); // returns "    Hello world"
```

Intersection Types

Intersection types are closely related to union types, but they are used very differently. An intersection type combines multiple types into one. This allows you to add together existing types to get a single type that has all the features you need. We use the singular AND operator “&” (bitwise AND in other languages) to intersect the wanted types.

```
interface IStudent {
  id: string;
  age: number;
}

interface IWorker {
  companyId: string;
}

type A = IStudent & IWorker;

let x: A;
x.age = 5;
x.companyId = 'CID5241';
x.id = 'ID3241';
```

Generics

In languages like C# and Java, one of the main tools in the toolbox for creating reusable components is generics, that is, being able to create a component that can work over a variety of types rather than a single one. This allows users to consume these components and use their own types.

For example, the identity function is a function that will return back whatever is passed in:

```
function identity<T>(arg: T): T {
  return arg;
}

// Explicitly set "T" to be string as one of the arguments to the
// function call
let output = identity<string>("myString");
//      ^ = let output: string

// type argument inference
// the compiler set the value of "T" for us automatically based on
// the type of the argument passed
let output = identity("myString");
//      ^ = let output: string
```

We've now added a type variable **T** to the identity function. This **T** allows us to capture the type the user provides (e.g. *number*), so that we can use that information later. Here, we use **T** again as the return type.

Utility types

TypeScript provides several utility types to facilitate common type transformations. These utilities are available globally. They can come in handy from time to time to shorten your code or clean up the logic of your types.

You can find all the available utility types and their description in the [TypeScript Handbook - Utility Types](#).

Working with Third-Party Types

As many solutions for the problems you are dealing with while developing an application are already dealt with, we tend to use third-party solutions in the form of libraries.

One of the problems you can and will run into, is using a third party library written in JavaScript in your TypeScript codebase and run into this:

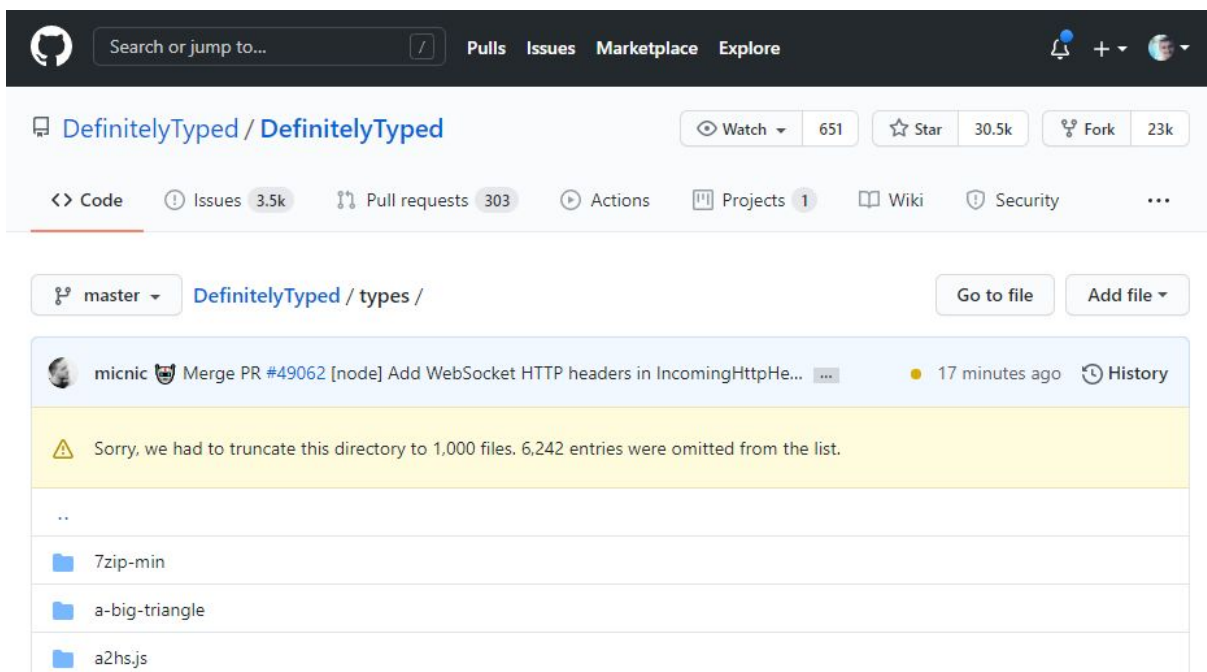
```
import React, { useMemo } from 'react';
import { ThemeProvider } from 'react-themeprovider';
import { BrowserRouter as Router } from 'react-router-dom';
import { ToastContainer, Slide } from 'react-toastify';
import { QueryCache, ReactQuery } from 'react-query';
import { ReactQueryDevtools } from 'react-query-devtools';
import ErrorBoundary from 'components/ErrorBoundary';

Could not find a declaration file for module 'react'.
f:/repos/inqool/vsb/web/node_modules/react/index.js implicitly has an 'any' type.
If the 'react' package actually exposes this module, consider sending a pull request to amend
'https://github.com/DefinitelyTyped/DefinitelyTyped/tree/master/types/react' ts(7016)
Peek Problem (Alt+F8) Quick Fix... (Ctrl+)
```

There are two ways for to deal with this problem:

1. Importing typings from **@types/[library_name]** (project DefinitelyTyped)

DefinitelyTyped is a massive GitHub repository that stores types for most JavaScript libraries. It's structured as a monorepo, so types for every compatible JavaScript library are present here. These types are published to the npm namespace **@types**, e.g. **@types/react** or **@types/styled-components**.

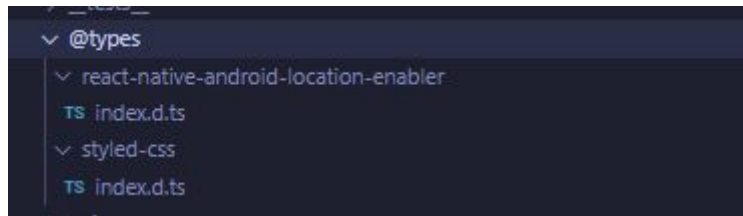


The screenshot shows the GitHub repository page for **DefinitelyTyped / DefinitelyTyped**. The repository has 651 watchers, 30.5k stars, and 23k forks. The current view is the **types** directory on the **master** branch. A recent merge pull request (PR #49062) is visible, titled "Merge PR #49062 [node] Add WebSocket HTTP headers in IncomingHttpHe...". A warning message indicates that the directory listing was truncated to 1,000 files, with 6,242 entries omitted. The visible files in the directory are:

- 7zip-min
- a-big-triangle
- a2hs.js

2. Define your own types

Creating our own types is pretty easy in TypeScript. By default, TypeScript will automatically pick up any type definitions that are inside the `@types` directory in `node_modules`, as well as any `index.d.ts` files. This means you can create an `index.d.ts` file wherever you want in your project, and TypeScript will use it. It is suggested, we organize the types inside the “@types” directory in our application:



Inside the `index.d.ts` file, you can declare all the types you want for that package. Make sure to declare the module or namespace that you want the types to be defined inside.

```
declare module 'react-native-android-location-enabler' {  
  You, 6 months ago | 1 author (You)  
  interface PromptFuncOptions {  
    // Set the desired interval for active Location updates, in milliseconds. Default: 10000  
    interval: number;  
    // Explicitly set the fastest interval for Location updates, in milliseconds. Default: 5000  
    fastInterval: number;  
  }  
  
  // The user has accepted to enable the location services  
  type PromptFuncResponse =  
    | 'already-enabled' // if the Location services has been already enabled  
    | 'enabled'; // if user has clicked on OK button in the popup  
  
  type PromptFunc = (opts: PromptFuncOptions) => Promise<PromptFuncResponse>;  
  
  You, 6 months ago | 1 author (You)  
  export default class RNAndroidLocationEnabler {  
    static promptForEnableLocationIfNeeded: PromptFunc;  
  }  
}
```

Resources

- TypeScript Documentation
 - <https://www.typescriptlang.org/docs/home.html>
- TypeScript Playground
 - <https://www.typescriptlang.org/play>
- React +TypeScript Cheatsheets
 - <https://github.com/typescript-cheatsheets/react-typescript-cheatsheet>
- JSON to TypeScript
 - <https://app.quicktype.io>
- TypeSearch (TS typings for JS libraries)
 - <https://microsoft.github.io/TypeSearch>
- Adding/Using TypeScript in create-react-app
 - <https://create-react-app.dev/docs/adding-typescript>