A collection of essential TypeScript types

Many of the types here should have been built-in. You can help by suggesting some of them to the TypeScript project.

Either add this package as a dependency or copy-paste the needed types. No credit required. 👌

PR welcome for additional commonly needed types and docs improvements. Read the contributing guidelines first.

Install

```
$ npm install type-fest
```

Requires TypeScript >=3.4

Usage

```
import {Except} from 'type-fest';

type Foo = {
    unicorn: string;
    rainbow: boolean;
};

type FooWithoutRainbow = Except<Foo, 'rainbow'>;
//=> {unicorn: string}
```

API

Click the type names for complete docs.

Basic

- `Primitive` - Matches any primitive value.
- `Class` - Matches a `class` constructor.
- `TypedArray` - Matches any typed array, like `Uint8Array` or `Float64Array`.
- `JsonObject` - Matches a JSON object.
- `JsonArray` - Matches a JSON array.
- `JsonValue` - Matches any valid JSON value.

- `ObservableLike` - Matches a value that is like an Observable.

Utilities

- `Except` - Create a type from an object type without certain keys. This is a stricter version of `Omit`.
- `Mutable` - Convert an object with `readonly` keys into a mutable object. The inverse of `Readonly<T>`.
- `Merge` - Merge two types into a new type. Keys of the second type overrides keys of the first type.
- `MergeExclusive` - Create a type that has mutually exclusive keys.
- `RequireAtLeastOne` - Create a type that requires at least one of the given keys.
- `RequireExactlyOne` - Create a type that requires exactly a single key of the given keys and disallows more.
- `PartialDeep` - Create a deeply optional version of another type. Use `Partial<T>` if you only need one level deep.
- `ReadonlyDeep` - Create a deeply immutable version of an `object` / `Map` / `Set` / `Array` type. Use `Readonly<T>` if you only need one level deep.
- `LiteralUnion` - Create a union type by combining primitive types and literal types without sacrificing auto-completion in IDEs for the literal type part of the union. Workaround for Microsoft/TypeScript#29729.
- `Promisable` - Create a type that represents either the value or the value wrapped in `PromiseLike`.
- `Opaque` - Create an opaque type.
- `SetOptional` - Create a type that makes the given keys optional.
- `SetRequired` - Create a type that makes the given keys required.
- `ValueOf` - Create a union of the given object's values, and optionally specify which keys to get the values from.
- `PromiseValue` - Returns the type that is wrapped inside a `Promise`.
- `AsyncReturnType` - Unwrap the return type of a function that returns a `Promise`.
- `ConditionalKeys` - Extract keys from a shape where values extend the given `Condition` type.
- `ConditionalPick` - Like `Pick` except it selects properties from a shape where the values extend the given `Condition` type.
- `ConditionalExcept` - Like `Omit` except it removes properties from a shape where the values extend the given `Condition` type.
- `UnionToIntersection` - Convert a union type to an intersection type.
- `Stringified` - Create a type with the keys of the given type changed to `string` type.
- `FixedLengthArray` - Create a type that represents an array of the given type and length.
- `IterableElement` - Get the element type of an `Iterable` / `AsyncIterable`. For example, an array or a generator.
- `Entry` - Create a type that represents the type of an entry of a collection.
- `Entries` - Create a type that represents the type of the entries of a collection.
- `SetReturnType` - Create a function type with a return type of your choice and the same parameters as the given function type.
- `Asyncify` - Create an async version of the given function type.

Template literal types

Note: These require TypeScript 4.1 or newer.

- `CamelCase` – Convert a string literal to camel-case (`fooBar`).
- `KebabCase` – Convert a string literal to kebab-case (`foo-bar`).
- `PascalCase` – Converts a string literal to pascal-case (`FooBar`)

- `SnakeCase` – Convert a string literal to snake-case (`foo_bar`).
- `DelimiterCase` – Convert a string literal to a custom string delimiter casing.

Miscellaneous

- `PackageJson` - Type for npm's `package.json` file.
- `TsConfigJson` - Type for TypeScript's `tsconfig.json` file (TypeScript 3.7).

Declined types

If we decline a type addition, we will make sure to document the better solution here.

- `Diff` and `Spread` - The PR author didn't provide any real-world use-cases and the PR went stale. If you think this type is useful, provide some real-world use-cases and we might reconsider.
- `Dictionary` - You only save a few characters (`Dictionary<number>` vs `Record<string, number>`) from `Record` , which is more flexible and well-known. Also, you shouldn't use an object as a dictionary. We have `Map` in JavaScript now.
- `SubType` - The type is powerful, but lacks good use-cases and is prone to misuse.
- `ExtractProperties` and `ExtractMethods` - The types violate the single responsibility principle. Instead, refine your types into more granular type hierarchies.

Tips

Built-in types

There are many advanced types most users don't know about.

- `Partial<T>` - Make all properties in `T` optional.

  ▶ Example
- `Required<T>` - Make all properties in `T` required.

  ▶ Example
- `Readonly<T>` - Make all properties in `T` readonly.

  ▶ Example
- `Pick<T, K>` - From `T` , pick a set of properties whose keys are in the union `K` .

  ▶ Example
- `Record<K, T>` - Construct a type with a set of properties `K` of type `T` .

  ▶ Example
- `Exclude<T, U>` - Exclude from `T` those types that are assignable to `U` .

  ▶ Example
- `Extract<T, U>` - Extract from `T` those types that are assignable to `U` .

  ▶ Example
- `NonNullable<T>` - Exclude `null` and `undefined` from `T` .

  ▶ Example
- `Parameters<T>` - Obtain the parameters of a function type in a tuple.

  ▶ Example

- `ConstructorParameters<T>` - Obtain the parameters of a constructor function type in a tuple.

  ▶ Example
- `ReturnType<T>` – Obtain the return type of a function type.

  ▶ Example
- `InstanceType<T>` – Obtain the instance type of a constructor function type.

  ▶ Example
- `Omit<T, K>` – Constructs a type by picking all properties from T and then removing K.

  ▶ Example

You can find some examples in the TypeScript docs.

Maintainers

- Sindre Sorhus
- Jarek Radosz
- Dimitri Benin
- Pelle Wessman

License

(MIT OR CC0-1.0)

Get professional support for this package with a Tidelift subscription

Tidelift helps make open source sustainable for maintainers while giving companies assurances about security, maintenance, and licensing for their dependencies.