Mar 21, 2024 · 12 min read

# A practical guide to TypeScript decorators

**Rahman Fadhil**
Developer and content writer.

## Table of contents

Decorators in JavaScript vs. TypeScript

Getting started with decorators in TypeScript

New TypeScript decorators

Types of decorators

- Class decorators
- Method decorators
- Property decorators
- Accessor decorators
- Auto-accessor decorators

Parameter decorators

Use cases for TypeScript decorators

Calculating execution time

***Editor's note:*** *This article was last updated by Yan Sun on 21 March 2024 to provide information about parameter decorators in TypeScript, and explore advanced decorator patterns like decorator composition and factories.*


A decorator is a programming design pattern in which you wrap something to change its behavior. This feature is currently at stage three in JavaScript. Decorators are not new; several programming languages, such as Python, Java, and C#, adopted this pattern before JavaScript. Further refinement of the syntax will require feedback from implementation and users.

At the time of writing, most browsers do not support decorators. Nonetheless, you

can test them out by using compilers like Babel.

In this article, we will learn how decorators compare in JavaScript and TypeScript. We will also explore the various types of TypeScript decorators, including the class, method, property, and accessor decorators.

## Decorators in JavaScript vs. TypeScript

TypeScript's decorator feature differs significantly from JavaScript's. The first big difference concerns what we can decorate. TypeScript decorators allow us to annotate and modify class declarations, methods, properties, accessors, and parameters.

TypeScript 5.0 introduces the new implementation of decorator support, which aligns with the ECMAScript stage three proposal. However, it does not currently support parameter decoration, but that might change in future ECMAScript proposals. Old TypeScript decorators do support parameter decoration. We'll learn more about this later in the article. JavaScript, on the other hand, only lets us decorate class declarations and methods.

The second important difference between decorators in JavaScript and TypeScript is type checking. Because TypeScript is a strongly typed programming language, it can type-check the parameters and return the value of the decorator function. JavaScript lacks this type checking and validation, so you need to rely on runtime checks or external tools like linters to catch type errors.

## Getting started with decorators in TypeScript

Start by creating a blank Node.js project:

```
$ mkdir typescript-decorators
$ cd typescript-decorators
```

```
$ npm init -y
```

Next, install TypeScript as a development dependency:

```
$ npm install -D typescript @types/node
```

The `@types/node` package contains the Node.js type definitions for TypeScript. We need this package to access some Node.js standard libraries.

Add an npm script in the `package.json` file to compile your TypeScript code:

```
{
  // ...
  "scripts": {
    "build": "tsc"
  }
}
```

Until TypeScript 5.0, we had to explicitly set a flag, `experimentalDecorators`, to use decorators in our code. With TypeScript 5.0, this is no longer the case. While such a flag will likely stay around for the foreseeable future, we can use new-style decorators without it. In fact, the old-style decorators modeled a different version of the proposal (Stage 2). We can use both styles in our code because the type rules differ, but it's not advisable.

Remember to configure your working environment to use at least TypeScript 5. Otherwise, the code in this article won't compile.

We'll use `ES6` as a target for TypeScript because all modern browsers support it:

```
{
  "compilerOptions": {
    "target": "ES6"
```

```
  }
}
```

Next, we'll create a simple TypeScript file to test the project out:

```
console.log("Hello, world!");


$ npm run build
$ node index.js
Hello, world!
```

Instead of repeating this command repeatedly, we can simplify the compilation and execution process by using a package called ts-node. It's a community package that enables us to run TypeScript code directly without compiling it first.

Let's install it as a development dependency:

```
$ npm install -D ts-node
```

Next, add a `start` script to the `package.json` file:

```
{
  "scripts": {
    "build": "tsc",
    "start": "ts-node index.ts"
  }
}
```

Simply run `npm start` to run your code:

```
$ npm start
Hello, world!
```

I have all the source code for this article published on my GitHub. You can clone it onto your machine using the command below:

```
$ git clone git@github.com:mdipirro/typescript-decorators.git
```

## New TypeScript decorators

In TypeScript, decorators are functions that can be attached to classes and their members, such as methods and properties.

In this section, we're going to look at new-style decorators. First, the new `Decorator` type is defined as follows:

```
type Decorator = (target: Input, context: {
  kind: string;
  name: string | symbol;
  access: {
    get?(): unknown;
    set?(value: unknown): void;
  };
  private?: boolean;
  static?: boolean;
  addInitializer?(initializer: () => void): void;
}) => Output | void;
```

The `type` definition above looks complex, so let's break it down one piece at a time:

- `target` represents the element we're decorating, whose type is `Input`
- `context` contains metadata about how the decorated method was declared, namely:
  - `kind` : The type of decorated value. As we'll see, this can be either `class` , `method` , `getter` , `setter` , `field` , or `accessor`

    - `name` : The name of the decorated object
    - `access` : An object with references to a getter and setter method to access the decorated object
    - `private` : Whether the decorated object is a `private` class member
    - `static` : Whether the decorated object is a `static` class member
    - `addInitializer` : A way to add custom initialization logic at the beginning of the constructor (or when the class is defined)
- `Output` represents the type of value returned by the `Decorator` function

In the next section, we'll examine the types of decorators. Interestingly, while old-style decorators let us decorate function parameters, new-style ones don't, at least for now. In fact, parameter decorators are waiting for a follow-on proposal to reach Stage 3.

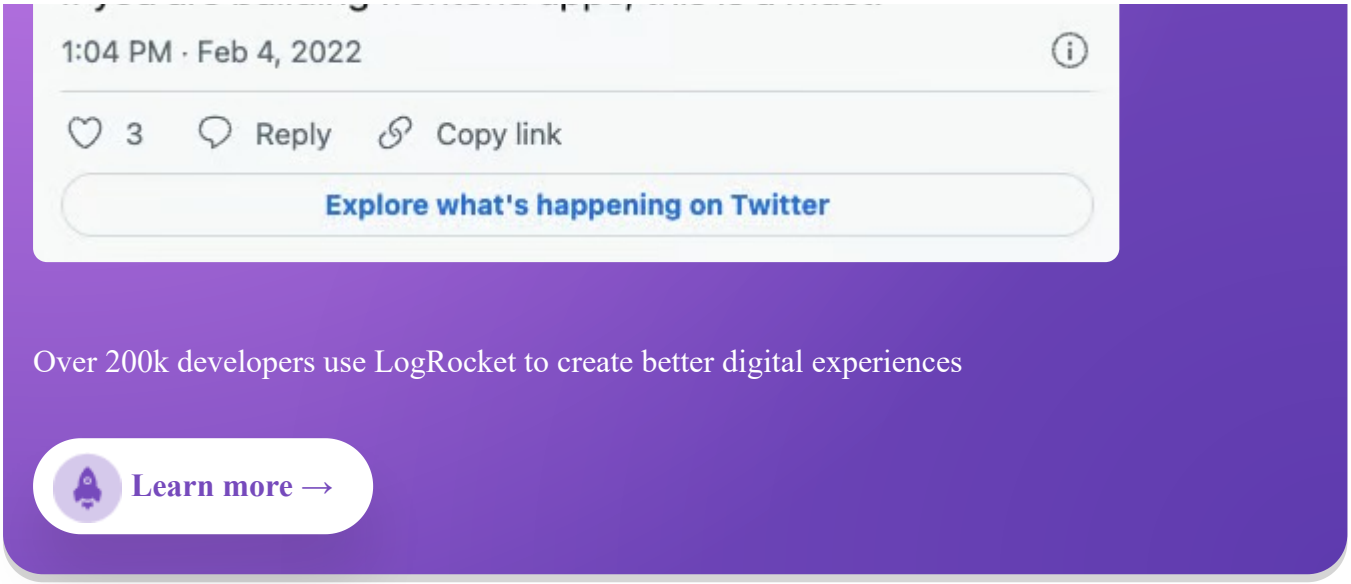
## Types of decorators

Now that we know how the Decorator type is defined, we'll examine the various types of decorators.

### Class decorators

Class decorators allow modification of class behavior. Upon class initialization, they are invoked and gain access to the class's constructor, methods, and properties.

---






---

When you attach a function to a class as a decorator, you'll receive the class constructor as the first parameter:

```
type ClassDecorator = (value: Function, context: {
  kind: "class"
  name: string | undefined
  addInitializer(initializer: () => void): void
}) => Function | void
```

For example, let's assume we want to use a decorator to add two properties, `fuel` and `isEmpty()`, to a `Rocket` class. In this case, we could write the following function:

```
function WithFuel(target: typeof Rocket, context): typeof Rocket {
  if (context.kind === "class") {
    return class extends target {
      fuel: number = 50
      isEmpty(): boolean {
        return this.fuel == 0
      }
    }
  }
```

```
}
```

After ensuring the `kind` of the decorated element is indeed `class`, we return a new class with two additional properties. Alternatively, we could have used prototype objects to add new methods dynamically:

```
function WithFuel(target: typeof Rocket, context): typeof Rocket {
  if (context.kind === "class") {
    target.prototype.fuel = 50
    target.prototype.isEmpty = (): boolean => {
      return this.fuel == 0
    }
  }
}
```

We can use `WithFuel` as follows:

```
@WithFuel
class Rocket {}

const rocket = new Rocket()
console.log((rocket as any).fuel)
console.log(`Is the rocket empty? ${(rocket as any).isEmpty()}`)
/* Prints:
50
Is the rocket empty? false
*/
```

You might have noticed that we had to cast `rocket` to `any` to access the new properties. That's because decorators can't influence the structure of the type.

If the original class defines a property that is later decorated, the decorator overrides the original value. For example, if `Rocket` has a `fuel` property with a different

value, `WithFuel` would override

such a value:

```
function WithFuel(target: typeof Rocket, context): typeof Rocket {
  if (context.kind === "class") {
    return class extends target {
      fuel: number = 50
      isEmpty(): boolean {
        return this.fuel == 0
      }
    }
  }
}
@WithFuel
class Rocket {
  fuel: number = 75
}

const rocket = new Rocket()
console.log((rocket as any).fuel)
// prints 50
```

Another use of the class decorators is to enhance the existing class methods. Let's say we have an `addFuel` method in the `Rocket` class:

```
class Rocket {
  fuel: number = 11;
  addFuel(amount: number) {
    this.fuel += amount;
  }
}
```

We can extend the `addFuel` method by applying a new class decorator: `logFuel`:

```
function logFuel(target: Function, context) {
  const original = target.prototype.addFuel;
  target.prototype.addFuel = function (message: string) {
    console.log(`Before adding fuel, total fuel: ${this.fuel}`);
    original.apply(this, arguments);
    console.log(`After adding fuel, total fuel: ${this.fuel}`);
  };
}
```

The above decorator adds logging functionality around the `addFuel` method of the class, allowing us to track changes in the fuel level before and after each fuel addition operation. Now, when we add it to the `Rocket` class as below, the change in fuel amount will be printed in the console:

```
@logFuel
class Rocket {
  fuel: number = 11;
  addFuel(amount: number) {
    this.fuel += amount;
  }
}
const rocket = new Rocket();
rocket.addFuel(10);
/*
Before adding fuel, total fuel: 11
After adding fuel, total fuel: 21
*/
```

## Method decorators

Another good place to attach a decorator is class methods. In this case, the type of the decorator function is as follows:

```
type ClassMethodDecorator = (target: Function, context: {
  kind: "method"
  name: string | symbol
  access: { get(): unknown }
  static: boolean
  private: boolean
  addInitializer(initializer: () => void): void
}) => Function | void
```

We can use method decorators when we want something to happen before or after the invocation of the method being decorated.

For example, during development, it might be useful to log the calls using a given method or verify pre/post-conditions before/after the call. Additionally, we can influence how the method is invoked, for example, by delaying its execution or limiting the number of calls within a given amount of time.

Finally, we can use method decorators to mark a method as deprecated, logging a message to warn the user and tell them which method to use instead:

```
function deprecatedMethod(target: Function, context) {
  if (context.kind === "method") {
    return function (...args: any[]) {
      console.log(`${context.name} is deprecated and will be removed
      return target.apply(this, args)
    }
  }
}
```

Again, the first parameter of the `deprecatedMethod` function is, in this case, the method we're decorating. After making sure it's indeed a method ( `context.kind === "method"` ), we return a new `function` that wraps the decorated method and logs a warning message before calling the actual method call.

We can then use our new decorator as follows:

```
@WithFuel
class Rocket {
  fuel: number = 75
  @deprecatedMethod
  isReadyForLaunch(): Boolean {
    return !(this as any).isEmpty()
  }
}

const rocket = new Rocket()
console.log(`Is the rocket ready for launch? ${rocket.isReadyForLaun
```

In the `isReadyForLaunch()` method, we refer to the `isEmpty` method we added via the `WithFuel` decorator. Notice how we had to cast `this` to an instance of `any`, as we did before. When we call `isReadyForLaunch()`, we'll see the following output, showing that the warning gets correctly printed out:

```
isReadyForLaunch is deprecated and will be removed in a future versi
Is the rocket ready for launch? true
```

Method decorators can be useful if you want to extend the functionality of our methods, which we'll cover later.

## Property decorators

Property decorators are very similar to method decorators:

```
type ClassPropertyDecorator = (target: undefined, context: {
  kind: "field"
```

```
  name: string | symbol
  access: { get(): unknown, set(value: unknown): void }
  static: boolean
  private: boolean
}) => (initialValue: unknown) => unknown | void
```

Not surprisingly, the use cases for property decorators are very similar to those for method decorators. For example, we can track the accesses to a property or mark it as deprecated:

```
function deprecatedProperty(_: any, context) {
  if (context.kind === "field") {
    return function (initialValue: any) {
      console.log(`${context.name} is deprecated and will be removed
      return initialValue
    }
  }
}
```

The code is very similar to the `deprecatedMethod` decorator we defined for methods, and so is its usage.

## Accessor decorators

Very similar to method decorators are accessor decorators, which are decorators that target getters and setters:

```
type ClassSetterDecorator = (target: Function, context: {
  kind: "setter"
  name: string | symbol
  access: { set(value: unknown): void }
  static: boolean
  private: boolean
```

```
  addInitializer(initializer: () => void): void
}) => Function | void

type ClassGetterDecorator = (value: Function, context: {
  kind: "getter"
  name: string | symbol
  access: { get(): unknown }
  static: boolean
  private: boolean
  addInitializer(initializer: () => void): void
}) => Function | void
```

The definition of accessor decorators is similar to that of method decorators. For example, we can merge our `deprecatedMethod` and `deprecatedProperty` decorations into a single, `deprecated` function that features support for getters and setters as well:

```
function deprecated(target, context) {
  const kind = context.kind
  const msg = `${context.name} is deprecated and will be removed in
  if (kind === "method" || kind === "getter" || kind === "setter") {
    return function (...args: any[]) {
      console.log(msg)
      return target.apply(this, args)
    }
  } else if (kind === "field") {
    return function (initialValue: any) {
      console.log(msg)
      return initialValue
    }
  }
}
```

## Auto-accessor decorators

The new decorator proposal also introduced a new element called the “auto-accessor field”:

```
class Test {
  accessor x: number
}
```

The transpiler will turn the `x` field above into a pair of getter and setter methods, with a `private` property behind the scenes. This is useful to represent a simple accessor pair and helps avoid some edgy issues that might arise while using decorators on class fields.

---

## More great articles from LogRocket:



---

Auto-accessors can also be decorated, and their type will essentially be a merge of

`ClassSetterDecorator` and `ClassGetterDecorator`. You can find additional details in the Stage 3 decorators pull request.

## Parameter decorators

Please note that the parameter decorator is not supported in TypeScript 5.0. The TypeScript documentation states: "This new decorator proposal is not compatible with — `emitDecoratorMetadata`, and it does not allow decorating parameters. Future ECMAScript proposals may be able to help bridge that gap."

If you are still supporting the older version of TypeScript code, here is how we can use the parameter decorator.

A parameter decorator is placed before the parameter's declaration and used to observe parameter declarations within methods. It takes three parameters:

- The constructor function of the class (for static) or the class prototype (for instance)
- The method name
- The ordinal index of the parameter

Below is a simple example:

```
function paramLogger(target: any, methodName: string, parameterIndex
  console.log(`Parameter ${parameterIndex + 1} of ${methodName} meth
}
class Rocket {
  launch(@paramLogger name: string) {
    console.log(`Launching ${name} in 3... 2... 1... 🚀`);
  }
}

const rocket = new Rocket();
rocket.launch('Space X');
```

```
/* output:
Parameter 1 of launch method
Launching Space X in 3... 2... 1... 🚀
*/
```

The above example defines a parameter decorator `paramLogger` that logs the index and name of a parameter. It's applied to the `name` parameter of `launch` method. When an instance of `Rocket` class is created and its `launch` method is invoked, we can observe the console output showing the logging of the parameter information.

# Use cases for TypeScript decorators

Now that we've covered what decorators are and how to use them properly, let's look at some specific problems they can help us solve.

## Calculating execution time

Let's say we want to estimate how long it takes to run a function to gauge your application performance. We can create a decorator to calculate the execution time of a method and print it on the console:

```
class Rocket {
  @measure
  launch() {
    console.log("Launching in 3... 2... 1... 🚀");
  }
}
```

The `Rocket` class has a `launch` method inside of it. To measure the execution time of the `launch` method, you can attach the `measure` decorator:

```
import { performance } from "perf_hooks";

function measure(target: Function, context) {
  if (context.kind === "method") {
    return function (...args: any[]) {
      const start = performance.now()
      const result = target.apply(this, args)
      const end = performance.now()

      console.log(`Execution time: ${end - start} milliseconds`)
      return result
    }
  }
}
```

As you can see, the `measure` decorator replaces the original method with a new one that enables it to calculate the execution time of the original method and log it to the console. To calculate the execution time, we'll use the Performance Hooks API from the Node.js standard library. Instantiate a new `Rocket` instance and call the `launch` method:

```
const rocket = new Rocket()
rocket.launch()
```

You'll get the following result:

```
Launching in 3... 2... 1... 🚀
Execution time: 1.062355000525713 milliseconds
```

## Advanced decorator patterns

In TypeScript, advanced decorator patterns such as decorator composition and

factories provide flexibility and reusability when applying decorators to classes, methods, and properties.

## Decorator composition

Decorator composition involves applying multiple decorators to a single class, method, or property, allowing for combining different behaviors to achieve more dynamic functionality. We can compose decorators by stacking them one after another, with each decorator executing sequentially.

Below is an example of two decorators: `minimumFuel` and `maximumFuel`:

```
function minimumFuel(fuel: number) {
  return function (target: Function, context) {
    if (context.kind === 'method') {
      return function (...args: any[]) {
        if (this.fuel > fuel) {
          console.log('fuel is more than mininum');
          return target.apply(this, args);
        } else {
          console.log(`Not enough fuel. Required: ${fuel}, got ${thi
        }
      };
    }
  };
}
function maximumFuel(fuel: number) {
  return function (target: Function, context) {
    if (context.kind === 'method') {
      return function (...args: any[]) {
        if (this.fuel < fuel) {
          console.log('fuel is less than maximum');
          return target.apply(this, args);
        } else {
```

```
            console.log(`excessive fuel. Maximum: ${fuel}, got ${this.
          }
        };
      }
    };
}
```

We can apply both decorators to the `launch` method, so both conditions can be enforced: the fuel level must be within the minimum and maximum levels before initiating the launch:

```
class Rocket {
  fuel: number = 11;
  @maximumFuel(100)
  @minimumFuel(10)
  launch() {
    console.log('Launching in 3... 2... 1... 🚀');
  }
}
const rocket = new Rocket();
rocket.launch();
/*
fuel is less than maximum
fuel is more than mininum
Launching in 3... 2... 1... 🚀
*/
```

When using multiple decorators in combination, the sequence is important. The decorators execute in the order they are listed. In this case, the `maximumFuel` decorator is applied first, followed by the `minimumFuel` decorator.

## Decorator factory

To configure our decorators to act differently in a certain scenario, we can use a concept called the decorator factory. Decorator factories are functions returning a decorator. This enables us to customize the behavior of our decorators by passing some parameters to the factory.

Take a look at the example below:

```
function fill(value: number) {
  return function(_, context) {
    if (context.kind === "field") {
      return function (initialValue: number) {
        return value + initialValue
      }
    }
  }
}
```

The `fill` decorator factory generates a decorator that can be applied to a property. When applied, it adds the value provided to the decorator to the value provided during field initialization:

```
class Rocket {
  @fill(20)
  fuel: number = 50
}
const rocket = new Rocket()
console.log(rocket.fuel) // 70
```

We can also use the decorator factory to create a class decorator. Here is an example of a class decorator called `Throttle`, which throttles the execution of the `launch` method:

```
function Throttle(delay: number) {
```

```
  return function (target: any, context: any) {
    const original = target.prototype.launch;
    let timeout = false;
    target.prototype.launch = function (message: string) {
      if (!timeout) {
        original.apply(this, arguments);
        timeout = true;
        setTimeout(() => {
          timeout = false;
        }, delay);
      } else {
        console.log(`Throttle, wait next time`);
      }
    };
  };
}

@Throttle(1000)
class Rocket {
  launch() {
    console.log('Launching in 3... 2... 1... 🚀');
  }
}
const rocket = new Rocket();
rocket.launch();
rocket.launch();
/* output
Launching in 3... 2... 1... 🚀
Throttle, wait next time
*/
```

In the above code, we retrieve the original method `launch` from the target class prototype, then call the `launch` method if the timeout is active; otherwise, we skip the call.

In this example, the `Rocket` class is decorated with the newly created decorator, specifying a throttle duration of 1000 milliseconds (one second). As shown in the console output, the first call to `launch` is successful, but the second call is skipped as it is still within the timeout period.

## Automatic error guard

Another common use case for decorators is checking pre- and post-conditions on method calls. For example, assume we want to make sure `fuel` is at least a given value before calling the `launch()` method:

```
class Rocket {
  fuel = 50

  launch() {
    console.log("Launching to Mars in 3... 2... 1... 🚀")
  }
}
```

Let's say we have a `Rocket` class that has a `launchToMars` method. To launch a rocket, the fuel level must be above, for example, 75.

Let's create the decorator for it:

```
function minimumFuel(fuel: number) {
  return function(target: Function, context) {
    if (context.kind === "method") {
        return function (...args: any[]) {
          if (this.fuel > fuel) {
            return target.apply(this, args)
          } else {
            console.log(`Not enough fuel. Required: ${fuel}, got ${t
          }
```

```
        }
      }
    }
  }
```

`minimumFuel` is a factory decorator. It takes the fuel parameter, indicating how much fuel is needed to launch a particular rocket. Just like in the previous use case, to check the fuel condition, wrap the original method with a new method. Notice how we can freely refer to `this.fuel`, which will only work at runtime.

Now, we can plug our decorator into the `launch` method and set the minimum fuel level:

```
class Rocket {
  fuel = 50

  @minimumFuel(75)
  launch() {
    console.log("Launching to Mars in 3... 2... 1... 🚀")
  }
}
```

If we now invoke the `launch` method, it won't launch the rocket because the current fuel level is 50:

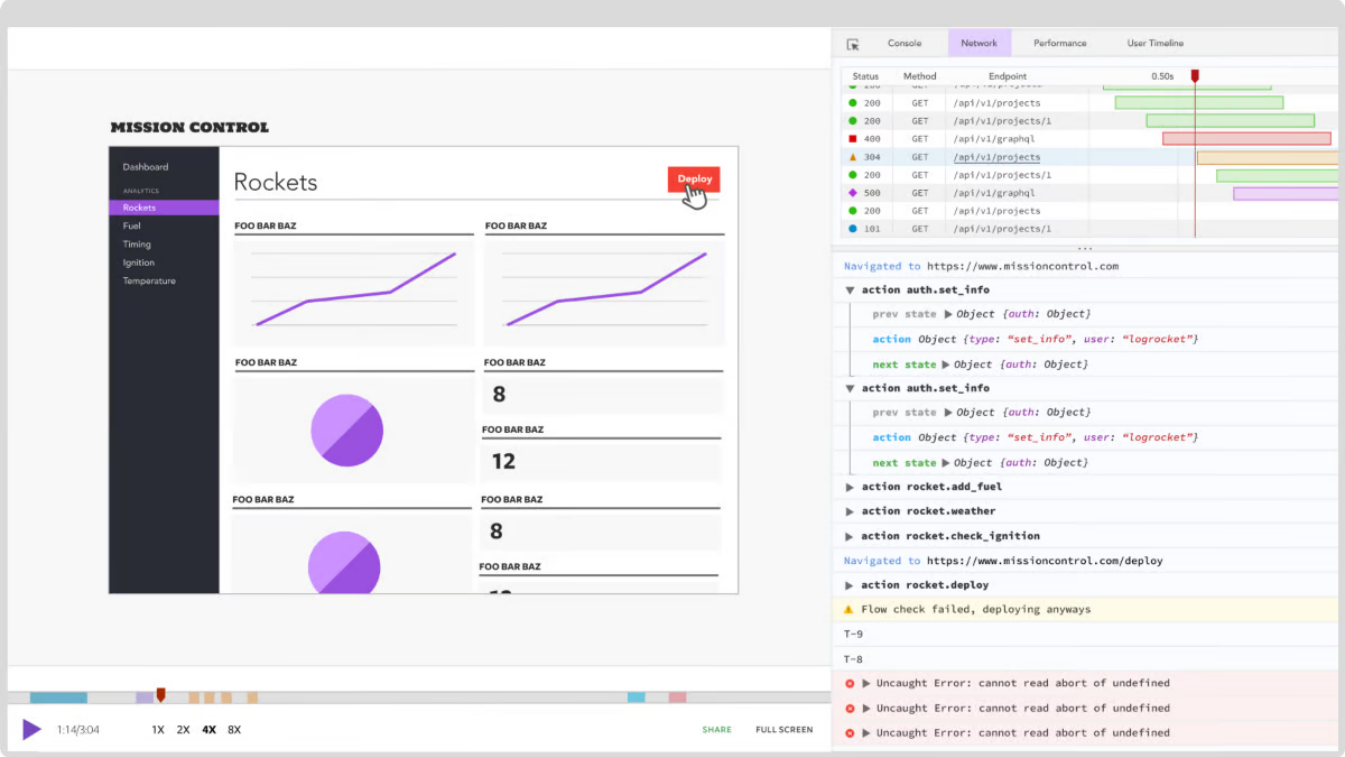
```
const rocket = new Rocket()
rocket.launch()

Not enough fuel. Required: 75, got 50
```

The cool thing about this decorator is that you can apply the same logic to a different method without rewriting the whole if-else statement.

## Conclusion

In some scenarios, we don't need to create custom decorators. Many TypeScript libraries and frameworks, such as TypeORM and Angular, already provide all the decorators we need. But it's always worth the extra effort to understand what's going on under the hood, and it might even inspire you to build your own TypeScript framework.

## LogRocket: Full visibility into your web and mobile apps




LogRocket is a frontend application monitoring solution that lets you replay problems as if they happened in your own browser. Instead of guessing why errors happen or asking users for screenshots and log dumps, LogRocket lets you replay the session to quickly understand what went wrong. It works perfectly with any app, regardless of framework, and has plugins to log additional context from Redux, Vuex,

and @ngrx/store.

In addition to logging Redux actions and state, LogRocket records console logs, JavaScript errors, stacktraces, network requests/responses with headers + bodies, browser metadata, and custom logs. It also instruments the DOM to record the HTML and CSS on the page, recreating pixel-perfect videos of even the most complex single-page and mobile apps.

Try it for free.

---

**Share this:**

#typescript



**Recent posts:**

## 7 TUI libraries for creating interactive terminal apps

When writing applications, a good user interface is just as important as the actual app's functionality. A good user interface [...]

**Yashodhan Joshi**
Jun 14, 2024 · 18 min read

## Expo Router adoption guide: Overview, examples, and alternatives

Expo Router provides an excellent file-based routing solution with crucial features such as deep linking and native support.

**Marie Starck**
Jun 13, 2024 · 8 min read

## Superglue vs. Hotwire for modern frontend development

Explore how Superglue and Hotwire revolutionize frontend development with HTML over the wire, enhancing performance, flexibility, and ease of use.

**Frank Joseph**

Jun 12, 2024 · 7 min read

## Using PocketBase to build a full-stack application

PocketBase is a performant Go-based tool that comes with

essential features like user auth, file uploads, access control rules, and more.

**Rahul Padalkar**
Jun 11, 2024 · 18 min read

---

View all posts

## 6 Replies to "A practical guide to TypeScript decorators"

**Oli** says: Reply

November 30, 2020 at 1:26 am

Thank you! Was looking for an explanation just like yours.

**Carlos** says: Reply

March 25, 2021 at 10:11 am

The class decorator example is wrong. The returned class does not have the “fuel” property. Please check this.

**Assad Isah** says: Reply

March 27, 2021 at 12:03 am

console.log((rocket).fuel)

**Tayyab** says: Reply

January 20, 2022 at 2:52 pm

Thank you, this was super helpful!

**khalvai** says:

Reply

March 20, 2022 at 10:09 am

thanks for your great and easy understanding content .

**X** says:

Reply

February 7, 2024 at 5:29 am

Is there any way to get it to change the type Rocket to have the properties, so you don't have to cast it to any?

## Leave a Reply