O que são Decorators e por que usar? Eles vão salvar a legibilidade do seu código!

Gustavo Oliveira
Software Engineer

Motivação - ECMAScript Decorators

- As classes ES6 são intencionalmente mínimas e não suportam alguns comportamentos comuns necessários de classes;

- Alguns casos requerem algum tipo de programabilidade ou introspecção. Decoradores tornam as declarações de classes programáveis;

- Atualmente são amplamente utilizados em JavaScript por meio de *transpilers*. Por exemplo: *core-decorators, ember-decorators, Angular, vue-property-decorator.*

ECMAScript Decorators ou JavaScript Decorators?

Ecma TC39

Ecma International, Technical Committee 39 - ECMAScript

The web https://www.ecma-international.org/memento/t...

Repositories 145 | Packages | People 80 | Projects

Pinned repositories

ecma262
Status, process, and documents for ECMA-262
HTML ★ 9.4k 748

proposals
Tracking ECMAScript Proposals
★ 9.2k 345

test262
Official ECMAScript Conformance Test Suite
JavaScript ★ 1.2k 292

agendas
TC39 meeting agendas
JavaScript ★ 522 138

ecma402
Status, process, and documents for ECMA 402
HTML ★ 256 69

notes
TC39 meeting notes
JavaScript ★ 35 3

Find a repository... Type: All Language: All

proposal-readonly-collections
Proposal: snapshot,diverge,readOnlyView methods for all collections
Apache-2.0 0 ★ 12 5 1 Updated 2 hours ago

Top languages
HTML JavaScript CSS
Shell

proposal-optional-chaining

HTML 64 4,520 12 1 Updated on Sep 27

proposal-decorators

Decorators for ES6 classes

HTML 57 965 73 (3 issues need help) 0 Updated on Sep 22

proposal-private-methods

Private methods and getter/setters for ES6 classes

HTML 22 226 3 1 Updated on Sep 17

proposal-private-fields

A Private Fields Proposal for ECMAScript

HTML 16 307 17 0 Updated on Sep 16

proposal-realms

ECMAScript Proposal, specs, and reference implementation for Realms

HTML 29 411 35 (1 issue needs help) 0 Updated on Aug 27

README.md

JavaScript Decorators

Stage 2

Status

Decorators are a JavaScript language feature, proposed for standardization at TC39. Decorators are currently at Stage 2 in TC39's process, indicating that the committee expects them to eventually be included in the standard JavaScript programming language. The decorators champion group is considering a redesign of the proposal as "static decorators", which the rest of this document describes.

The idea of this proposal

This decorators proposal aims to improve on past proposals by working towards twin goals:

- It should be easy not just to use decorators, but also to write your own.
- Decorators should be fast, both generating good code in transpilers, and executing fast in native JS implementations.

This proposal enables the basic functionality of the JavaScript original decorators proposal (e.g., most of what is available in TypeScript decorators), as well as two additional capabilities of the previous Stage 2 proposal which were especially important: access to private fields and methods, and registering callbacks which are called during the constructor.

stage	name	mission
0	strawman	Present a new feature (proposal) to TC39 committee. Generally presented by TC39 member or TC39 contributor.
1	proposal	Define use cases for the proposal, dependencies, challenges, demos, polyfills etc. A champion (TC39 member) will be responsible for this proposal.
2	draft	This is the initial version of the feature that will be eventually added. Hence description and syntax of feature should be presented. A transpiler such as Babel should support and demonstrate implementation.
3	candidate	Proposal is almost ready and some changes can be made in response to critical issues raised by adopters and TC39 committee.
4	finished	The proposal is ready to be included in the standard.

https://tc39.es/process-document

Então como usar Decorator se ainda não faz parte da especificação?

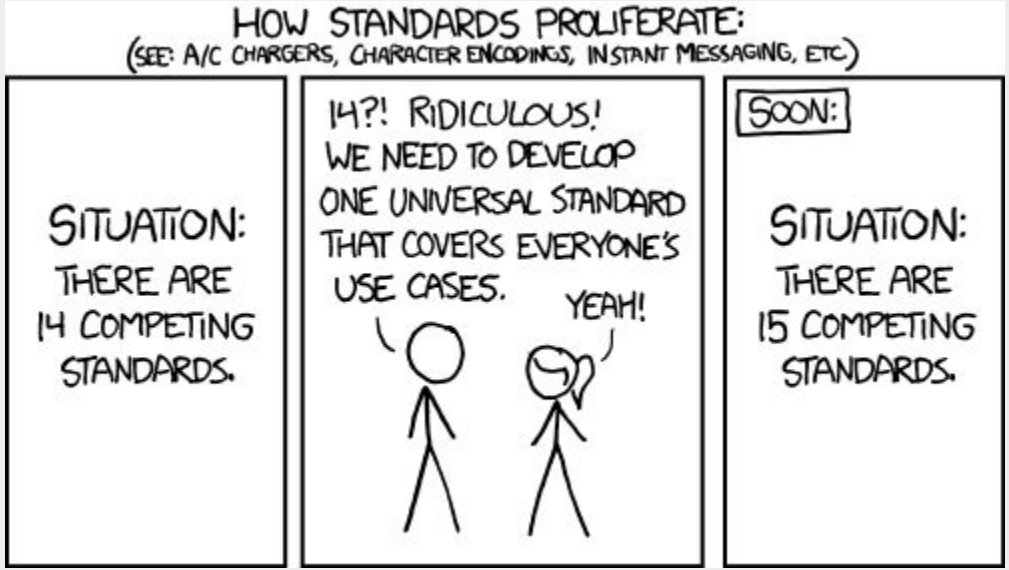
Usar um *transpile* como TypeScript ou Babel!

Usaremos Babel com a extensão:

@babel/plugin-proposal-decorators

Um pouco de história

- Babel introduziu Decorators (*stage 1*) na versão 5, mas eles foram removidos na versão 6 porque a proposta ainda estava em andamento.
- Voltou na versão 7 após uma reescrita completa
- Ainda pode usar o stage 1 setando a *flag legacy* para *true*



Para entender Decorators, precisamos primeiro entender o que é ***property descriptor***


A PALESTRA É SOBRE DECORATORS
MAS É PRECISO SABER ALGUNS FUNDAMENTOS ANTES
imgflip.com

Property descriptor

É um conjunto de regras do objeto, as regras são:

- Value - É o valor da propriedade;
- Writable - Se o *value* da propriedade pode ser mudado;
- Enumerable - Se essa propriedade será exibida em enumerações, como o *loop - for in* ou o *loop - for of* ou *Object.keys* etc.;
- Configurable - Se alguma propriedade do Descritor pode ser alterada
- Getters and Setters.

Property descriptor - Get Descriptor

```
var myObj = {
    myPropOne: 1,
    myPropTwo: 2
};

let descriptor = Object.getOwnPropertyDescriptor(
    myObj, 'myPropOne'
);

console.log(descriptor);
```

```
> \
> node index.js
{ value: 1, writable: true, enumerable: true,
 configurable: true }
```

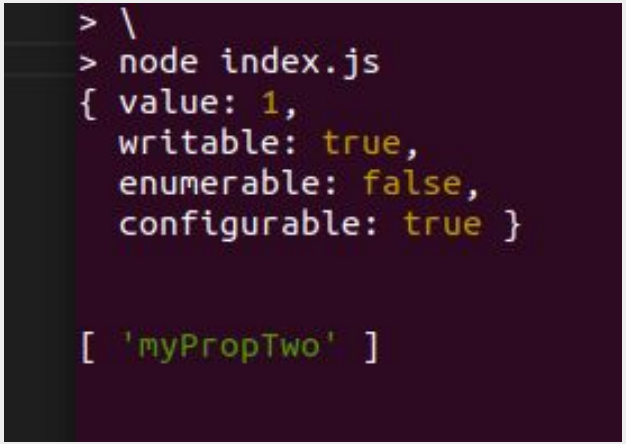
Property descriptor - Analisando a regra *whitable*

```
'use strict';

var myObj = {
    myPropOne: 1,
    myPropTwo: 2
};

// modificando a propriedade
Object.defineProperty(myObj, 'myPropOne', {
    writable: false
});

// printando a propriedade
let descriptor = Object.getOwnPropertyDescriptor(
    myObj, 'myPropOne'
);
console.log(descriptor);

// setando novo valor
myObj.myPropOne = 2
```

```
> node index.js
{ value: 1,
  writable: false,
  enumerable: true,
  configurable: true }
/home/gustavo-de-o-feitosa/workspace/tdc/decorator/index.js:2
myObj.myPropOne = 2
                ^

TypeError: Cannot assign to read only property 'myPropOne' of
    at Object.<anonymous> (/home/gustavo-de-o-feitosa/workspa
    at Module._compile (internal/modules/cjs/loader.js:778:30
    at Object.Module._extensions..js (internal/modules/cjs/lo
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:
    at Function.Module.runMain (internal/modules/cjs/loader.j
    at startup (internal/bootstrap/node.js:283:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:622:3)
```

Property descriptor - Analisando a regra *enumerable*

```
// modificando descriptor
Object.defineProperty(myObj, 'myPropOne', {
    enumerable: false
});

// printando descriptor
let descriptor = Object.getOwnPropertyDescriptor(
    myObj, 'myPropOne'
);
console.log(descriptor);

// printando keys
console.log(
    Object.keys(myObj)
);
```



Property descriptor - Analisando a regra configurable

```
// modificando descriptor
Object.defineProperty(myObj, 'myPropThree', {
    value: 3,
    writable: false,
    configurable: false,
    enumerable: true
});

// printando descriptor
let descriptor = Object.getOwnPropertyDescriptor(
    myObj, 'myPropThree'
);
console.log(descriptor);

// mudando descriptor
Object.defineProperty(myObj, 'myPropThree', {
    writable: true
});
```

```
> node index.js
{ value: 3,
  writable: false,
  enumerable: true,
  configurable: false }
/home/gustavo-de-o-feitosa/workspace/tdc/decorator/index.js:2
Object.defineProperty(myObj, 'myPropThree', {
       ^

TypeError: Cannot redefine property: myPropThree
    at Function.defineProperty (<anonymous>)
    at Object.<anonymous> (/home/gustavo-de-o-feitosa/workspa
    at Module._compile (internal/modules/cjs/loader.js:778:30
    at Object.Module._extensions..js (internal/modules/cjs/lo
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:
    at Function.Module.runMain (internal/modules/cjs/loader.j
    at startup (internal/bootstrap/node.js:283:19)
    at bootstrapNodeJSCore (internal/bootstrap/node.js:622:3)
```

Property descriptor - Getters and Setters

```
var fuel = 'petrol';
var car = {
  name: 'SuperFast',
  maker: 'Ferrari',
  engine: 'v12'
};

Object.defineProperty(car, 'engineDetails', {
  get: function () {
    return fuel + " " + this.engine + " engine";
  },
  set: function (details) {
    let splits = details.split(' ');
    fuel = splits[0];
    this.engine = splits[1];
  }
});

console.log(car.engineDetails);
car.engineDetails = "diesel v8";
console.log(car.engineDetails);
```

```
> \
> \
> node index.js


petrol v12 engine


diesel v8 engine
```

Problema

Como criar um sistema de *Log* para ser ser acionado sempre que um determinado método for chamado?

Exemplo

```
class myClass {
  myMethod() { }
}
myClass.prototype.myMethod = myFunc(myClass.prototype.myMethod);
```

```
console.log(`Iniciando função ${nomeFunc} com argumentos ${args} às ${new Date()}.`);
```

Funciona? Funciona!

```
class myClass {
    myMethod(arg) {
        const date = (new Date).toString()
        console.log(`Iniciando função ${'myMethod'} com argumento ${arg} em ${date}.`);

        ...
        ...
        ...
    }
}
```

Mas e se tivessem mais métodos?

Bom, seria uma bagunça!

```
class myClass {
    myMethod(arg) {
        const date = (new Date).toString()
        console.log(`Iniciando função ${'myMethod'} com argumento ${arg} em ${date}.`);
        ...
    }
    myMethod2(arg) {
        const date = (new Date).toString()
        console.log(`Iniciando função ${'myMethod2'} com argumento ${arg} em ${date}.`);
        ...
    }
    ...
    ...
    ...
    myMethodN(arg) {
        const date = (new Date).toString()
        console.log(`Iniciando função ${'myMethod3'} com argumento ${arg} em ${date}.`);
        ...
    }
}
```

Melhorou… Mas ainda ta ruim!

```
const log = (nameMethod, arg) => {
  const date = (new Date).toString()
  console.log(`Iniciando função ${nameMethod} com argumento ${arg} em ${date}.`)
}

class myClass {
  myMethod(arg) {
    log('myMethod', arg)
    ...
  }
  myMethod2(arg) {
    log('myMethod2', arg)
    ...
  }
  ...
  ...
  ...
  myMethodN(arg) {
    log('myMethodN', arg)
    ...
  }
}
```

Decorators

Decorators

- A proposta do estágio 2 incentiva os desenvolvedores a escreverem seus próprios Decorators. Para isso haverá um conjunto de decorators predefinidos que servem como base:

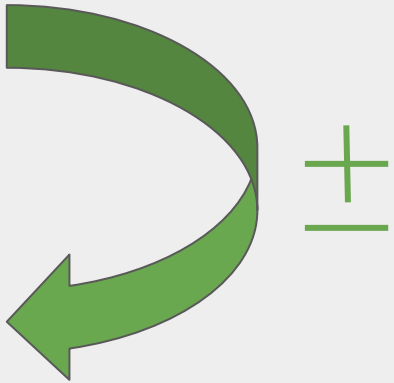
    - @wrap: Substitui um método ou a classe inteira pelo valor de retorno de uma determinada função;
    - @register: Chama um *callback* após a criação da classe;
    - @expose: Chama um *callback* de uma função passada para acessar campos ou métodos privados após a criação da classe;
    - @initialize: Executa um determinado *callback* ao criar uma instância da classe.

Decorators

- A proposta do estágio 2 incentiva os desenvolvedores a escreverem seus próprios Decorators. Para isso haverá um conjunto de decorators predefinidos que servem como base:
    - @wrap: Substitui um método ou a classe inteira pelo valor de retorno de uma determinada função;
    - @register: Chama um *callback* após a criação da classe;
    - @expose: Chama um *callback* de uma função passada para acessar campos ou métodos privados após a criação da classe;
    - @initialize: Executa um determinado *callback* ao criar uma instância da classe.

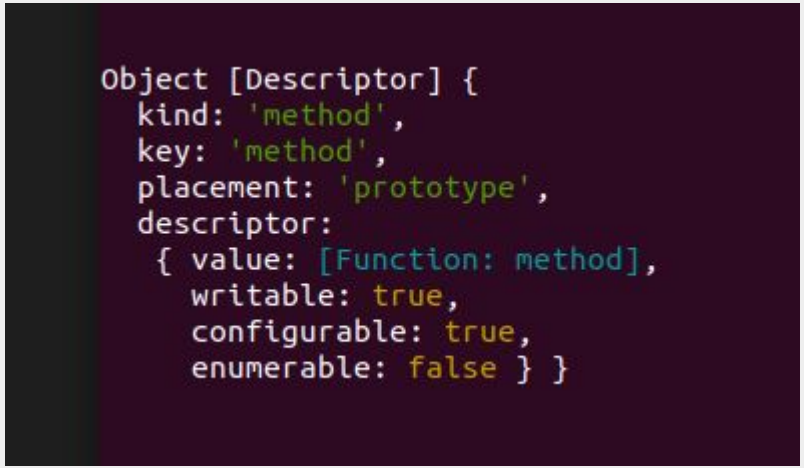
Decorators

```
class myClass {
  @wrap(myFunc)
  myMethod() { }
}

// Equivalente mais ou menos à:

class myClass {
  myMethod() { }
}
myClass.prototype.myMethod = myFunc(myClass.prototype.myMethod);
```



Decorators - Vamos testar!

```
const logger = wrapped ⇒ {
    console.log(wrapped)
}

class myClass {
    @logger
    myMethod(arg) {
        return arg
    }
}

const test = new myClass()

test.myMethod('arg')
```



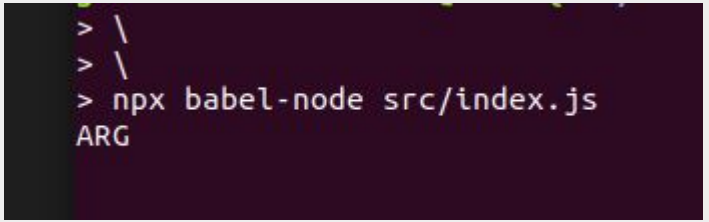
```
const toUpperCase = str => str.toUpperCase()

const logger = wrapped => {
  const { descriptor } = wrapped
  const originalFunc = descriptor.value

  descriptor.value = originalFunc
}

class myClass {
  @logger
  myMethod(arg) {
    return arg + ' test'
  }
}

const myInstance = new myClass()

const res = myInstance.myMethod('arg')

console.log(res)
```

Testando...

```
> \
> \
> npx babel-node src/index.js
arg test
```

```
const toUpperCase = str => str.toUpperCase()

const logger = wrapped => {
  const { descriptor } = wrapped
  const originalFunc = descriptor.value

  descriptor.value = toUpperCase
}

class myClass {
  @logger
  myMethod(arg) {
    return arg + ' test'
  }
}

const myInstance = new myClass()

const res = myInstance.myMethod('arg')

console.log(res)
```

Mudando *myMethod*



Escrevendo o log

```
const loggerFunc = originalFunc => arg => {
  const date = new Date().toString()
  const nameFunc = originalFunc.name

  console.log(`Iniciando função "${nameFunc}"\
com argumento "${arg}" em ${date}.`)

  return originalFunc(arg)
}

const logger = wrapped => {
  const { descriptor } = wrapped
  const originalFunc = descriptor.value

  descriptor.value = loggerFunc(originalFunc)
}

class myClass {
  @logger
  myMethod(arg) {
    return arg + ' test'
  }
}

const myInstance = new myClass()

const res = myInstance.myMethod('arg')

console.log(res)
```

Sem Decorators x com Decorators

```
const log = (nameMethod, arg) ⇒ {
  const date = (new Date).toString()
  console.log(`Iniciando função ${nameMethod} com argumento ${arg} em ${date}.`)
}

class myClass {
  myMethod(arg) {
    log('myMethod', arg)
    ...
  }
  myMethod2(arg) {
    log('myMethod2', arg)
    ...
  }
  ...
  ...
  ...
  myMethodN(arg) {
    log('myMethodN', arg)
    ...
  }
}
```

```
class myClass {
  @logger
  myMethod(arg) {
    ...
  }
  @logger
  myMethod2(arg) {
    ...
  }
  ...
  ...
  ...
  @logger
  myMethodN(arg) {
    ...
  }
}
```

L*og* é propriedade de todo método!

L*og* é propriedade de todo método!

Então da para estender para a classe toda?

```
const logger = wrapped => {
  console.log(wrapped)
}

@logger
class myClass {
  myMethod(arg) {
    return arg + ' test'
  }
  myMethod2(arg) {
    return arg + ' test 2'
  }
}

const myInstance = new myClass()

const res = myInstance.myMethod('arg')
```

Aplicando em todos métodos da classe

```
> \
> \
> npx babel-node src/index.js

Object [Descriptor] {
  kind: 'class',
  elements:
   [ Object [Descriptor] {
       kind: 'method',
       key: 'myMethod',
       placement: 'prototype',
       descriptor: [Object] },
     Object [Descriptor] {
       kind: 'method',
       key: 'myMethod2',
       placement: 'prototype',
       descriptor: [Object] } ] }
```

O mesmo de antes...

```
const loggerFunc = originalFunc ⇒ arg ⇒ {
  const date = new Date().toString()
  const nameFunc = originalFunc.name

  console.log(`Iniciando função "${nameFunc}"\
com argumento "${arg}" em ${date}.`)

  return originalFunc(arg)
}

const loggerMethod = wrapped ⇒ {
  const { descriptor } = wrapped
  const originalFunc = descriptor.value

  descriptor.value = loggerFunc(originalFunc)
}
```

```
const loggerClass = wrapped => {
  const { elements } = wrapped

  elements.map(elemWrapped => {
    loggerMethod(elemWrapped)
  })
}

const logger = wrapped => {
  const { kind } = wrapped

  if (kind === 'method') loggerMethod(wrapped)
  else if (kind === 'class') loggerClass(wrapped)
}

@logger
class myClass {
  myMethod(arg) {
    return arg + ' test'
  }
  myMethod2(arg) {
    return arg + ' test 2'
  }
}

const myInstance = new myClass()

const res = myInstance.myMethod('arg')
const res2 = myInstance.myMethod2('arg')

console.log(res)
console.log(res2)
```

```
Iniciando função "myMethod" com argumento "arg" em Sat
Nov 23 2019 09:09:54 GMT-0300 (Brasilia Standard Time).

Iniciando função "myMethod2" com argumento "arg" em Sat
 Nov 23 2019 09:09:54 GMT-0300 (Brasilia Standard Time)
.



arg test

arg test 2
```

Dá pra fazer algo a mais por enquanto?

Métodos imutáveis

```
const readonlyMethod = wrapped => {
  const { descriptor } = wrapped

  descriptor.writable = false
}

const readonly = wrapped => {
  const { elements } = wrapped

  elements.map(elemWrapped => {
    readonlyMethod(elemWrapped)
  })
}

@readonly
class myClass {
  myMethod(arg) {
    return arg
  }
  myMethod2(arg) {
    return arg
  }
}

myClass.prototype.myMethod = (arg, arg2) => {
  return arg + arg2
}
```

```
myClass.prototype.myMethod = function (arg) {
                             ^

TypeError: Cannot assign to read only property 'myMethod' of object
    at Object.<anonymous> (/home/gustavo-de-o-feitosa/workspace/tdc/
    at Module._compile (internal/modules/cjs/loader.js:778:30)
    at Module._compile (/home/gustavo-de-o-feitosa/workspace/tdc/dec
.js:99:24)
    at Module._extensions..js (internal/modules/cjs/loader.js:789:10
    at Object.newLoader [as .js] (/home/gustavo-de-o-feitosa/workspa
/lib/index.js:104:7)
    at Module.load (internal/modules/cjs/loader.js:653:32)
    at tryModuleLoad (internal/modules/cjs/loader.js:593:12)
    at Function.Module._load (internal/modules/cjs/loader.js:585:3)
    at Function.Module.runMain (internal/modules/cjs/loader.js:831:1
    at Object.<anonymous> (/home/gustavo-de-o-feitosa/workspace/tdc/
b/_babel-node.js:234:23)
```

O que esperar do futuro de Decorators em JS?

O que esperar do futuro de Decorators em JS?

- tc39/proposal-class-fields
- tc39/proposal-private-methods

tc39/proposal-class-fields

Private fields

The above example has some implementation details exposed to the world that might be better kept internal. Using ESnext private fields and methods, the definition can be refined to:

```
class Counter extends HTMLElement {
  #x = 0;

  clicked() {
    this.#x++;
    window.requestAnimationFrame(this.render.bind(this));
  }

  constructor() {
    super();
    this.onclick = this.clicked.bind(this);
  }

  connectedCallback() { this.render(); }

  render() {
    this.textContent = this.#x.toString();
  }
}
window.customElements.define('num-counter', Counter);
```

tc39/proposal-private-methods

Private methods and fields

The above example has some implementation details exposed to the world that might be better kept internal. Using ESnext private fields and methods, the definition can be refined to:

```
class Counter extends HTMLElement {
  #xValue = 0;

  get #x() { return #xValue; }
  set #x(value) {
    this.#xValue = value;
    window.requestAnimationFrame(this.#render.bind(this));
  }

  #clicked() {
    this.#x++;
  }

  constructor() {
    super();
    this.onclick = this.#clicked.bind(this);
  }

  connectedCallback() { this.#render(); }

  #render() {
    this.textContent = this.#x.toString();
  }
}
window.customElements.define('num-counter', Counter);
```

Na comunidade de python já é realidade a um bom tempo

```
@app.route("/add_tweet")
@login_required
def add_tweet():
    ...
```

```
@app.route("/admin/delete_user")
@login_required
@admin_login_required
def admin_delete_user():
    ...
```

```
from functools import wraps

def login_required(f):
    @wraps(f)
    def wrap(*args, **kwargs):

        # if user is not logged in, redirect to login page
        if not request.headers["authorization"]:
            return redirect("login page")

        # get user via some ORM system
        user = User.get(request.headers["authorization"])

        # make user available down the pipeline via flask.g
        g.user = user

        # finally call f. f() now haves access to g.user
        return f(*args, **kwargs)

    return wrap
```

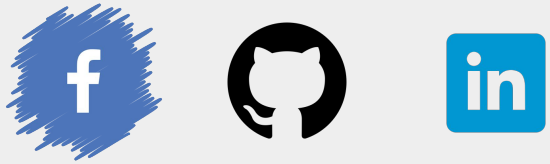
Dúvidas?

https://github.com/gustavooliveiraf/decorators

Fim!

Obrigado!

Gustavo Oliveira



gustavooliveiraf

Referências

- https://github.com/gustavooliveiraf/decorators
- https://github.com/tc39/proposal-decorators
- https://github.com/tc39/proposal-class-fields
- https://github.com/tc39/proposal-private-methods
- https://babeljs.io/docs/en/babel-plugin-proposal-decorators
- https://codeburst.io/javascript-object-property-attributes-ac012be317e2