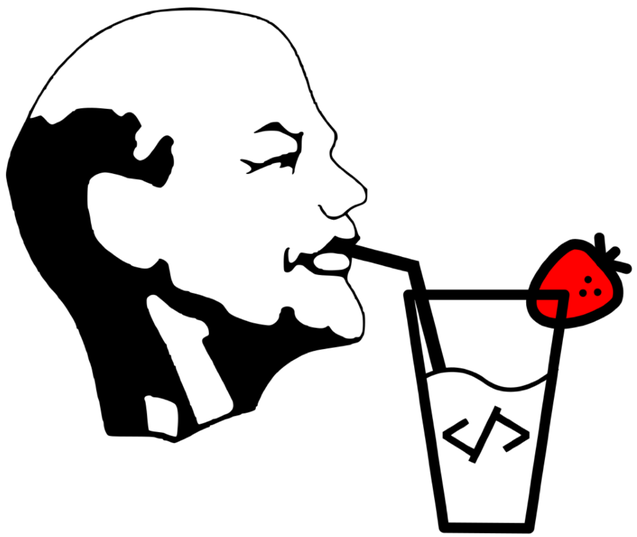
Reflection in TypeScript

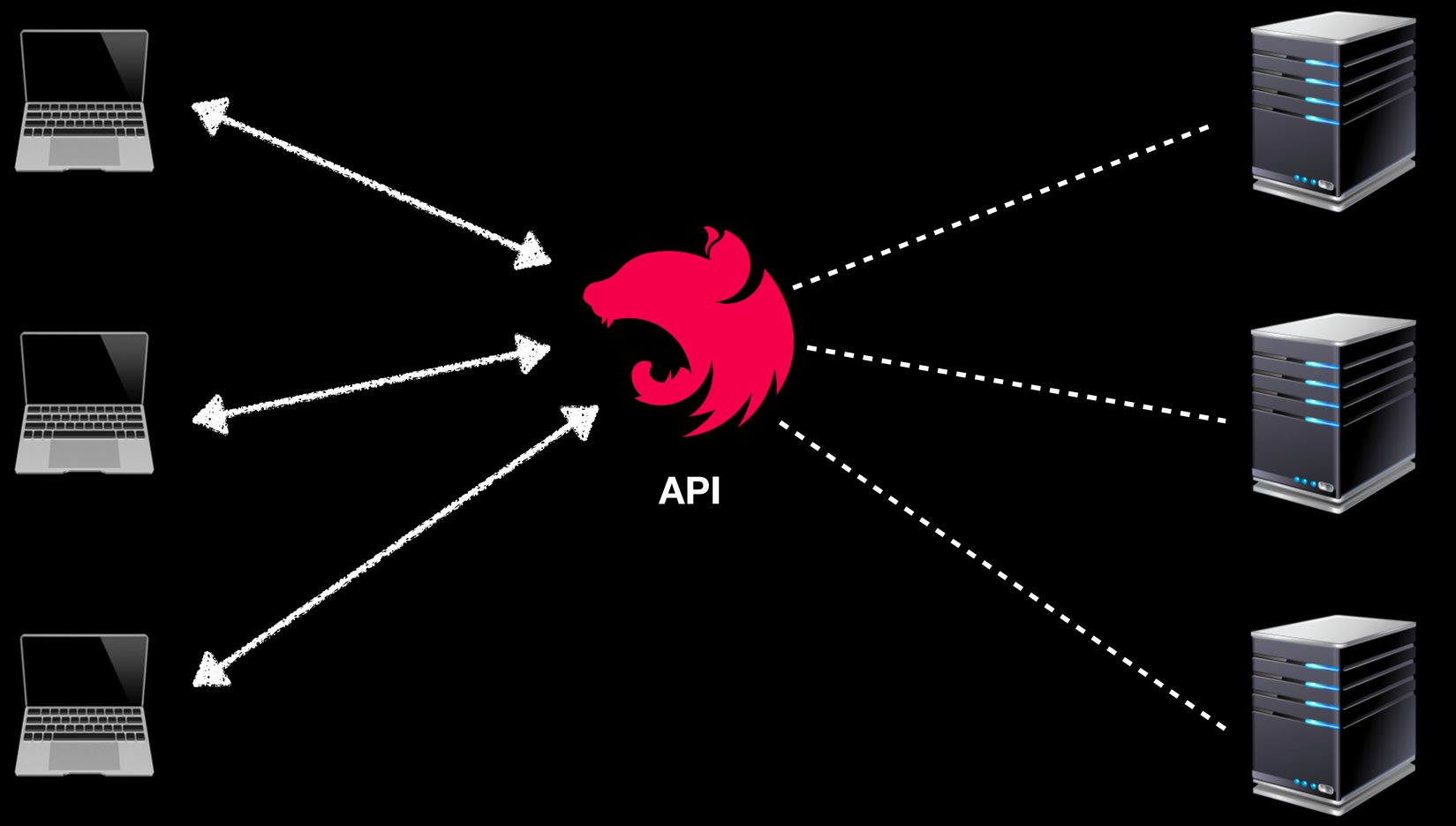
Alexander Bogachev


ARRIVAL
Welcome to Generation 2 electric vehicles
Marylebone
E
F



Фронтенд Юность

Problem



Role-based access control (RBAC)

https://habr.com/ru/company/custis/blog/248649/

https://habr.com/ru/company/custis/blog/258861/

Requirements

- Easy to use
- Store roles and permissions in our code with exact structure and connection between roles and permissions for audit demonstration.
- W/o dumb checks like IS_ADMIN
- Be sure all checks are performed
- Copy-paste protection (do yo really write each controller method?)
- Post-checks data from backend
- Good typings

Requirements 2

- Minimum of code while developing
- Minimum of boilerplate while using
- Maximum functionality
- Extensibility
- Concrete confidence of stable work

OpenSource solutions

- Complicated configuration
- Difficult to use
- Freaky roles storage (ini files 😿)
- No/bad types

nest-access-control (accesscontrol)

```
export enum AppRoles {
  USER_CREATE_ANY_VIDEO = 'USER_CREATE_ANY_VIDEO',
  ADMIN_UPDATE_OWN_VIDEO = 'ADMIN_UPDATE_OWN_VIDEO',
}

export const roles: RolesBuilder = new RolesBuilder();

roles
  .grant(AppRoles.USER_CREATE_ANY_VIDEO)
  .createOwn('video')
  .deleteOwn('video')
  .readAny('video')
  .grant(AppRoles.ADMIN_UPDATE_OWN_VIDEO)
  .extend(AppRoles.USER_CREATE_ANY_VIDEO)
  .updateAny('video', ['title'])
  .deleteAny('video');
```

```
import { roles } from './app.roles';

@Module({
  imports: [AccessControlModule.forRoles(roles)],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

```
@Controller()
export class AppController  {
    constructor(private readonly appService: AppService)  {}
    @UseGuards(AuthGuard, ACGuard)
    @UseRoles({
        resource:  'video',
        action:  'read',
        possession:  'any',
    })
    @Get()
    root(@UserRoles() userRoles: any)  {
        return this.appService.root(userRoles);
    }
}
```

nest-access-control (accesscontrol)

```
import {RolesBuilder} from 'nest-access-control';

export const roles: RolesBuilder = new RolesBuilder();

export enum AppRoles {
    USER_CREATE_ANY_VIDEO = 'USER_CREATE_ANY_VIDEO',
    ADMIN_UPDATE_OWN_VIDEO = 'ADMIN_UPDATE_OWN_VIDEO',
  }

roles.grant(AppRoles.ADMIN_UPDATE_OWN_VIDEO).
```

```
attributes                    (method) Access.attri…
create
createAny
createOwn
delete
deleteAny
deleteOwn
denied
deny
extend
grant
inherit
```

nestjs-acl (accesscontrol)

```
export const roles = new AccessControl({
    ADMIN: {
        doSomething: {
            'create:any': ['*']
        },
        doSomethingElse: {
            'create:any': ['*']
        }
    },
    USER: {
        doSomething: {
            'create:own': ['*']
        }
    }
});
```

```
export const userCanDoSomething = (opts) => {
    const {
        context: { user },
        data, // the data passed to the test
        sourceData // source data passed to the test
    } = opts;
    return [
        // rule 1
        {
            req: opts.rolesBuilder.can(opts.context.user.roles).createOwn('doSomething'),
            // do an extra check if with context
            check: () => opts.data.user === context.user
        },
        // rule 2
        {
            req: opts.rolesBuilder.can(opts.context.user.roles).createAny('doSomething')
        }
    ]
};
```

```
@Module({
    imports: [AclModule.register(roles), MyProvider]
})
export class MyModule {
    construtor(protected acl: AclService) {
        // register acl rules creators
        this.acl
            .registerRules('userCanDoSomething', userCanDoSomething)
            .registerRules('userCanDoSomethingElse', userCanDoSomethingElse);
    }
}
```

```
export class MyProvider {
    constructor(protected acl: AclService) {}

    /**
     * This method is protected by a rule with id userCanDoSomething
     */
    // this method is protected by a rule with id userCanDoSomething
    async doSomething(user: IRoles<string>) {
        const data = { foo: 'bar', user };
        // The AclService will throw a ForbiddenException if the check fails.
        const { rule, data } = await this.acl.check({
            id: 'userCanDoSomething',
            data,
            context: {
                user: {
                    roles: user.roles
                }
            }
        });
        // do something...
    }
}
```

nestjs-rbac

```
export const RBACstorage: IStorageRbac = {
    roles: ['admin', 'user'],
    permissions: {
      permission1: ['create', 'update', 'delete'],
      permission2: ['create', 'update', 'delete'],
      permission3: ['filter1', 'filter2', RBAC_REQUEST_FILTER],
      permission4: ['create', 'update', 'delete'],
    },
    grants: {
      admin: [
        '&user',
        'permission1',
        'permission3',
      ],
      user: ['permission2', 'permission1@create'],
    },
    filters: {
      filter1: TestFilterOne,
      filter2: TestFilterTwo,
      [RBAC_REQUEST_FILTER]: RequestFilter,
    },
  };
```

```
@Module({
    imports: [
      RBAcModule.forRoot(IStorageRbac),
    ],
    controllers: []
  })
  export class AppModule {}
```

```
@Controller()
export class RbacTestController {

  @RBAcPermissions('permission', 'permission@create')
  @UseGuards(
    GuardIsForAddingUserToRequestGuard,
    RBAcGuard,
  )
  @Get('/')
  async test1(): Promise<boolean> {
    return true;
  }
}
```

```typescript
@RBAcPermissions('permission', 'permission@create')
@UseGuards(
  GuardIsForAddingUserToRequestGuard,
  RBAcGuard,
)
@Get('/')
async test1(): Promise<boolean> {
  return true;
}
```

```typescript
@Get('/')
async getVehicles(
    @Policy() policy: ReadVehicleListPolicy
): Promise<boolean> {
    return true;
}
```

Content

- Reflection. Reflection in JS and TS
- Decorators. Combine well with Reflection
- Nest.js, reflection and decorators in Nest
- Our solution of RBAC

Reflection

- `Reflect.apply(target, thisArgument, argumentList)`
- `Reflect.construct(target, argumentList[, newTarget])`
- `Reflect.defineProperty(target, propertyKey, attributes)`
- `Reflect.deleteProperty(target, propertyKey)`
- `Reflect.get(target, propertyKey[, receiver])`
- `Reflect.ownKeys(target)`
- ...

Proxy

```
let obj = {
    a: 'someString'
};

obj = new Proxy(obj, {
    get(target, key) {
        if (key in target) {
            return target[key]
        }
        return 'defaultValue';
    }
});

obj.a // someString
obj.b // defaultValue
```

```
const obj = {
    _a: 'someString',
    get a() {
        return this._a;
    }
};

const proxyObj = new Proxy(obj, {
    get(target, key) {
        return target[key];
    }
})


const obj2 = {
    __proto__: proxyObj,
    _a: 'newString'
}

obj2.a // someString
```

```
const obj = {
    _a: 'someString',
    get a() {
        return this._a;
    }
};

const proxyObj = new Proxy(obj, {
    get(target, key, receiver) {
        return Reflect.get(target, key, receiver);
    }
})

const obj2 = {
    __proto__: proxyObj,
    _a: 'newString'
}

obj2.a // newString
```

JS / TS

Reflection in TS

Decorators

&

TC
39

https://babeljs.io/blog/2018/09/17/decorators

Old

```
class MyClass {
    @getDecorators().methods[name]
    foo() {}

    @decorator
    [bar]() {}
  }
```

```
const myObj = {
    @dec1 foo: 3,
    @dec2 bar() {},
  };
```

New

```
class MyClass {
    @decorator
    @dec(arg1, arg2)
    @namespace.decorator
    @(complex ? dec1 : dec2)
    method() {}
  }
```

BABEL

TS

```
function log2(target, propertyName, descriptor) {

}
```

Class decorator

Class constructor

```
function classDecorator<T extends Function>(target: T): T
{

}


@classDecorator
class MyClass {

}
```

Method decorator

```
function methodDecorator(
    target: Object,
    propertyKey: string | symbol,
    descriptor: TypedPropertyDescriptor<any>
): TypedPropertyDescriptor<any> {
{

}


class MyClass {
    @methodDecorator
    myMethod() {

    }
}
```

Class prototype

'myMethod'

```
{
  writeable: true,
  enumerable: true,
  configurable: true
}
```

Static method decorator

```
function methodDecorator(
    target: Function,
    propertyKey: string | symbol,
    descriptor: TypedPropertyDescriptor<any>
): TypedPropertyDescriptor<any> {
{

}


class MyClass {
    @methodDecorator
    static myMethod() {

    }
}
```

Class constructor

'myMethod'

```
{
  writeable: true,
  enumerable: true,
  configurable: true
}
```

Param decorator

Class prototype

‘myMethod’

0

```
function paramDecorator(
    target: Object,
    propertyKey: string | symbol,
    index: number
): void { }



class MyClass {
    myMethod(
        @paramDecorator param
    ) {

    }
}
```

Field decorator

Class prototype

```
function fieldDecorator(
    target: Object,
    propertyKey: string | symbol
): TypedPropertyDescriptor<any>
{

}
```

'myField'

```
class MyClass {
    @fieldDecorator
    myField: number;
}
```

Static field decorator

Class constructor

```
function staticFieldDecorator(
    target: Function,
    propertyKey: string | symbol
): TypedPropertyDescriptor<any>
{

}
```

'myField'

```
class MyClass {
    @staticFieldDecorator
    static myField: number;
}
```

tsconfig.json

```
{
    "compilerOptions": {
        "module": "commonjs",
        "declaration": true,
        "removeComments": true,
        // 👇
        "experimentalDecorators": true,
        "emitDecoratorMetadata": true
        // 👆
        // ...
    }
}
```

@Decorators

```
function f() {
    console.log('f(): evaluated');

    return function (target, propertyKey, descriptor) {
        console.log('f(): called');
    }
  }

function g() {
    console.log('g(): evaluated');

    return function (target, propertyKey, descriptor) {
        console.log('g(): called’);
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

```
function f() {
    console.log("f(): evaluated");
    return function (target, propertyKey, descriptor) {
        console.log("f(): called");
    }
  }

function g() {
    console.log("g(): evaluated");
    return function (target, propertyKey, descriptor) {
        console.log("g(): called");
    }
}

class C {
    @f()
    @g()
    method() {}
}
```

```
Init-time:
  f(): evaluated
  g(): evaluated
Run-time
  g(): called
  f(): called
```

```
function log(text) {
    return function(target, name, descriptor) {
        console.log(text);
    }
}

function log2(target, name, descriptor) {
    console.log(name);
}

class MyClass {
    @log('Log myMethod')
    @log2
    myMethod() {
        return 'my-method';
    }
}
```

???

Decorator Factories

Decorator

```
function log2(target, propName, descriptor) {
    // On method call
}
```

Factory

Decorator

```
function log(value: string) {
    // On app init
    return function(target, propName, descriptor) {
        // On method call
    }
}
```

```
class MyClass {
    @log('Log myMethod')
    @log2
    myMethod() {
        return 'my-method';
    }
}
```

How about types?

reflect-metadata

reflect-metadata

--emitDecoratorMetadata

reflect-metadata

babel-plugin-transform-typescript-metadata

How to use

Build: npx tsc test.reflect.ts --experimentalDecorators --emitDecoratorMetadata

Run: npx ts-node -r tsconfig-paths/register test.reflect.ts

Quokka.js

Try it

https://codesandbox.io/s/friendly-pascal-4kd0g

```
class Class {}


interface IClass {
    value: number;
}


class Demo {
    @logParamTypes
    doSomething(
        param1: string,
        param2: number,
        param3: Class,
        param4: IClass,
        param5: (a: number) => void
    ): number {
        return 1;
    }
}
```

```
function logParamTypes(target, key) {
    const types = Reflect.getMetadata(
        "design:paramtypes",
        target,
        key
    );

    const names = types.map(a => a.name).join();
    console.log(`${key} param types: ${names}`);

}
```

```
...

class Demo {
    @logParamTypes
    doSomething(
        param1: string,
        param2: number,
        param3: Class,
        param4: iClass,
        param5: (a: number) => void
    ): number {
        return 1;
    }
}

new Demo().doSomething(...);


// doSomething param types: String,Number,Class,Object,Function
```

```
...

class Demo {
    @logParamTypes
    doSomething(
        param1: string,
        param2: number,
        param3: Class,
        param4: iClass,
        param5: (a: number) => void
    ): number {
        return 1;
    }
}

new Demo().doSomething(...);
```

```
[Function: String]
function String() { [native code] }

[Function: Number]
function Number() { [native code] }

[Function: Class]
class Class {}

[Function: Object]
function Object() { [native code] }

[Function: Function]
function Function() { [native code] }
```

```
// doSomething param types: String,Number,Class,Object,Function
```

```
...

class Demo {
    @logParamTypes
    doSomething(
        param1: string,
        param2: number,
        param3: Class,
        param4: iClass,
        param5: (a: number) => void
    ): number {
        return 1;
    }
}

new Demo().doSomething(...);
```

```
[Function: String]
function String() { [native code] }

[Function: Number]
function Number() { [native code] }

[Function: Class]
class Class {}

[Function: Object]
function Object() { [native code] }

[Function: Function]
function Function() { [native code] }
```

```
// doSomething param types: String,Number,Class,Object,Function
```

design:paramtypes

design:type

design:returntype

```
function logReturnType(target, key) {
    const types = Reflect.getMetadata(
        "design:returntype",
        target,
        key
    );

    console.log(types);
}

class Demo {
    @logReturnType
    doSomething(): number {
        return 1;
    }
}

new Demo().doSomething();


// [λ: Number]
```

Own solution


Relax 🙂

nest-access-control

🔮

@Policy

👮🏽‍♀️

BasePolicy

👁️

PermissionInterceptor

```
export enum ROLE {
    USER = 'USER',
    VAN_DRIVER = ‘VAN_DRIVER',
    BUS_DRIVER = 'BUS_DRIVER',
    ADMIN = ‘ADMIN'
}
```

```
export enum PERMISSION {
    WORKSHIFT_START = 'WORKSHIFT_START',

    VEHICLE_READ = 'VEHICLE_READ',
    VEHICLE_UPDATE = 'VEHICLE_UPDATE',

}
```

```
export const rolePermissions = {
    [ROLE. BUS_DRIVER]: [
        PERMISSION.WORKSHIFT_START,
        PERMISSION.VEHICLE_READ,
}
```

```
@Controller()
@UseInterceptors(PermissionInterceptor)
export class VehicleController {

    @Get('/v1/vehicles')
    @UseAuthGuard()
    async getVehicles(
        @User() user: IUser,
        @Policy() policy: ReadVehicleListPolicy
    ): Promise<VehicleExtendedResponse[]> {

        const vehicleAccesses = await this.vehicleService.getAccesses({
            userId: user.id
        });

        if (!vehicleAccesses.length) {
            policy.skip();
            return [];
        }

        const vehicles = await this.vehicleService.getVehicles({
            ids: vehicleAccesses.map(va => va.vehicleId)
        });

        policy.ensure(vehicles, vehicleAccesses);

        return await this.prepareVehicles(vehicles, include);
    }
}
```

```
@Controller()
@UseInterceptors(PermissionInterceptor)
export class VehicleController {

    @Get('/v1/vehicles')
    @UseAuthGuard()
    async getVehicles(
        @User() user: IUser,
        @Policy() policy: ReadVehicleListPolicy
    ): Promise<VehicleExtendedResponse[]> {

        const vehicleAccesses = await this.vehicleService.getAccesses({
            userId: user.id
        });

        if (!vehicleAccesses.length) {
            policy.skip();
            return [];
        }

        const vehicles = await this.vehicleService.getVehicles({
            ids: vehicleAccesses.map(va => va.vehicleId)
        });

        policy.ensure(vehicles, vehicleAccesses);

        return await this.prepareVehicles(vehicles, include);
    }
}
```

BasePolicy

```
export abstract class BasePolicy {
 constructor(request: Request, permissions?: RawRolePermissions) {
    if (request) {
      this.request = request;
      this.permissions = getPreparedPermissionSet(permissions);
      this.checkAccess();
    }
  }

  public abstract get permission(): string;

  protected resolve(result = false): void {
    if (result !== true) {
      throw ENSURE_FAIL_403;
    }
    // Request update;
  }
  public fail(): void {
    throw PERMISSION_DENIED_403;
  }
  public skip(): void {
    this.resolve(true);
  }
}
```

ReadVehicleListPolicy

```
class ReadVehicleListPolicy extends BasePolicy {
    public get permission(): PERMISSION {
        return PERMISSION.VEHICLE_LIST_READ;
    }

    ensure(vehicles: IVehicle[], vehicleAccesses: IVehicleAccess[]): void {
        return super.resolve(
            vehicles.every(vehicle ⇒
                vehicleAccesses.some(
                    vehicleAccess ⇒ vehicleAccess.vehicleId ═══ vehicle.id &&
                    vehicleAccess.userId ═══ this.user.id
                )
            )
        );
    }

}
```

```
@Controller()
@UseInterceptors(PermissionInterceptor)
export class VehicleController {

    @Get('/v1/vehicles')
    @UseAuthGuard()
    async getVehicles(
        @User() user: IUser,
        @Policy() policy: ReadVehicleListPolicy
    ): Promise<VehicleExtendedResponse[]> {

        const vehicleAccesses = await this.vehicleService.getAccesses({
            userId: user.id
        });

        if (!vehicleAccesses.length) {
            policy.skip();
            return [];
        }

        const vehicles = await this.vehicleService.getVehicles({
            ids: vehicleAccesses.map(va => va.vehicleId)
        });

        policy.ensure(vehicles, vehicleAccesses);

        return await this.prepareVehicles(vehicles, include);
    }
}
```

@Policy

```
import 'reflect-metadata';

import { BasePolicy } from './BasePolicy';

const PERMISSION_METADATA = '__routeArguments__';
const CUSTOM_PERMISSION_METADATA = '__customRouteArgs__';

const usedPermissions = new Map<string, string>();
```

@Policy

```
export const Policy = (): ParameterDecorator => (
  target: Record<string, Function>,
  propertyName: string,
  index: number
): void => {

  const types = Reflect.getMetadata('design:paramtypes', target, propertyName);

  const __class = types[index];

  const metadata = {
    [`${CUSTOM_PERMISSION_METADATA}:${index}`]: {
      index,
      data: undefined,
      pipes: [],
      factory: (_: undefined, req: Request): BasePolicy => {
        return new __class(req);
      }
    }
  };

  Reflect.defineMetadata(PERMISSION_METADATA, metadata, target.constructor, propertyName);
};
```

@Policy

```
export const Policy = (): ParameterDecorator => (
  target: Record<string, Function>,
  propertyName: string,
  index: number
): void => {
  const types = Reflect.getMetadata('design:paramtypes', target, propertyName);

  const __class = types[index];
  const policy: BasePolicy = new __class();
  const permission = policy.permission;
  const usedPermission = usedPermissions.get(permission);

  if (usedPermission && usedPermission !== __class.name) {
    throw new Error(`Permission '${permission}' already used`);
  }
  usedPermissions.set(permission, __class.name);

  const metadata = {
    [`${CUSTOM_PERMISSION_METADATA}:${index}`]: {
      index,
      data: undefined,
      pipes: [],
      factory: (_: undefined, req: Request): BasePolicy => {
        return new __class(req);
      }
    }
  };

  Reflect.defineMetadata(PERMISSION_METADATA, metadata, target.constructor, propertyName);
};
```

```
const metadata = {
  [`${CUSTOM_PERMISSION_METADATA}:${index}`]: {
    index,
    data: undefined,
    pipes: [],
    factory: (_: undefined, req: Request): BasePolicy => {
      return new __class(req);
    }
  }
};

Reflect.defineMetadata(PERMISSION_METADATA, metadata, target.constructor, propertyName);
```

createParamDecorator()

```
@Controller()
@UseInterceptors(PermissionInterceptor)
export class VehicleController {

    @Get('/v1/vehicles')
    @UseAuthGuard()
    async getVehicles(
        @User() user: IUser,
        @Policy() policy: ReadVehicleListPolicy
    ): Promise<VehicleExtendedResponse[]> {

        const accesses = await this.vehicleService.getAccesses({
            userId: user.id
        });

        if (!vehicleAccesses.length) {
            policy.skip();
            return [];
        }

        const vehicles = await this.vehicleService.getVehicles({
            ids: vehicleAccesses.map(va => va.vehicleId)
        });

        policy.ensure(vehicles, vehicleAccesses);

        return await this.prepareVehicles(vehicles, include);
    }
}
```

PermissionInterceptor

```
@Injectable()
export class PermissionInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next.handle().pipe(
      tap(() => {
        const req = context.switchToHttp().getRequest();
        if (!req.permissionsChecks) {
          throw NO_PERMISSIONS_CHECK_500;
        }

        const isFalseChecks = Object.values(req.permissionsChecks).some(p => !p);
        if (isFalseChecks) {
          throw NO_PERMISSIONS_CHECK_500;
        }
      })
    );
  }
}
```

Requirements

- Easy to use
- Store roles and permissions in our code with exact structure and connection between roles and permissions for audit demonstration.
- W/o dumb checks like IS_ADMIN
- Be sure all checks are performed
- Copy-paste protection
- Post-checks data from backend
- Pretty types

Requirements 2

- Minimum of code while developing
- Minimum of boilerplate while using
- Maximum functionality
- Extensibility
- Concrete confidence of stable work

Who uses?

8,149 Dependents

Nest.js

```
@Controller()
@UseInterceptors(PermissionInterceptor)
export class CompanyController {
    @Get('/v1/company/:companyId/settings')
    @UseAuthGuard()
    @AdpApiOperation({
        title: `Gets company's settings`
    })
    @ApiImplicitParam({
        name: 'companyId',
        type: String,
        required: true,
        description: `Company's ID`
    })
    @ApiOkResponse({
        description: `Returns company's settings`,
        type: CompanySettingsResponse
    })
    async getCompanySettings(
        @Param() { companyId }: GetCompanySettingsParams,
        @User() user: IUser,
    ) {

    }
}
```

TypeORM

```
import {
    Entity, Column, PrimaryGeneratedColumn,
    OneToOne, JoinColumn
} from "typeorm";
import { Photo } from "./Photo";

@Entity()
export class PhotoMetadata {

    @PrimaryGeneratedColumn()
    id: number;

    @Column()
    comment: string;

    @OneToOne(type => Photo)
    @JoinColumn()
    photo: Photo;
}
```

class-validator

```
export class GetCompanySettingsParams {
    @IsUUID()
    @ApiModelProperty({
        description: `Company's ID`,
        example: UUID,
        type: String
    })
    readonly companyId: string;
}
```

nestjs/common

```
async getCompanySettings(
    @Param() { companyId }: GetCompanySettingsParams
): Promise<CompanySettingsResponse> {
    // ...
}
```

class-transformer

```
import { Expose } from 'class-transformer';

export class Invite {
    @Expose()
    id: string;

    @Expose()
    phone: string;

    createdAt: string;

    constructor(partial: Partial<IInvite>) {
        return {
            ...this,
            ...partial
        };
    }
}
```

class-transformer

```
import { Invite } from './models';

@Controller()
export class InviteController {
    constructor(private readonly authService: AuthService) {}

    @Get('/v1/invites/:id')
    async getInvite(@Param() dto: GetInviteByIdDto): Promise<Invite> {
        const invite = await this.authService.getInvite(dto.id);
        return new Invite(invite);
    }
}
```

MobX

```
class Timer {
    @observable start = Date.now()
    @observable current = Date.now()

    @computed
    get elapsedTime() {
        return this.current - this.start + "milliseconds"
    }

    @action
    tick() {
        this.current = Date.now()
    }
}
```

Angular

```
import { Component, HostListener } from '@angular/core';

@Component({
  selector: 'example-component',
  template: 'Woo a component!'
})
export class ExampleComponent {
  @HostListener('click', ['$event'])
  onHostClick(event: Event) {
    // clicked, `event` available
  }
}
```

autobind-decorator

```
class Component {
    constructor() {
      this.method = this.method.bind(this);
    }

    method() {
      return this.value;
    }
  }
```

```
class Component {
  @boundMethod
  method() {
    return this.value
  }
}
```

TS + @ + reflect-metadata = ❤️

t.me/rm_baad

twitter.com/rm_baad

https://github.com/rmbaad/holyjs-piter-2020