# Implementing Advanced Decorators


**Ivan Mushketyk**

@mushketyk | ivanm.io

# Types of Decorators

Methods

Classes

Accessors

Properties

Auto-accessors

Parameters *

# Decorators Based Framework

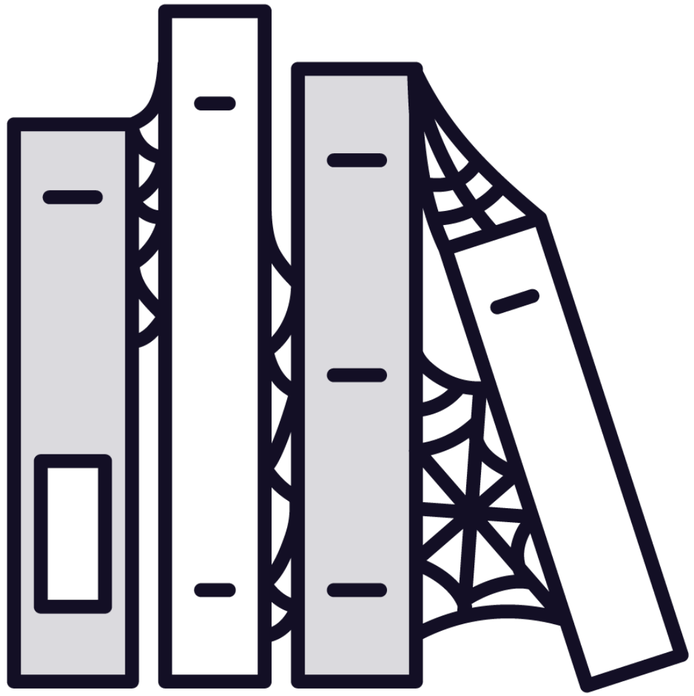
Web framework

Implement code to handle HTTP requests

Dependency injection

Create a graph of an object automatically

# Legacy Decorators



**Learn how to use experimental decorators**

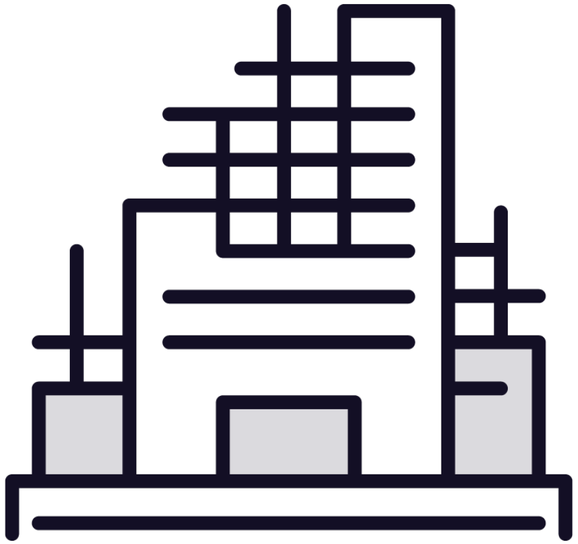
- Decorators for methods, classes, etc.
- Parameter decorators

**Why?**

- Legacy decorators are still used
- Have some advantages

**Re-implement web framework using experimental decorators**

# What Are We Going to Implement?



**A web framework**

- Heavily based on decorators
- Loosely inspired by Nest.js

**Features**

- Implement HTTP APIs
- Dependency injection
  - Implicitly create a graph of objects

# Impelmenting HTTP API

## Web framework for TypeScript

app.js

```
@Controller('/api')
class WeatherController {
  @Get('/weather')
  public get() {
    return {
      apiVersion: 'v1',
      temperature: 20,
      humidity: 80,
      city: 'London',
    }
  }
}
startApp()
```

```
> wget
weather.io/api/weather
{
  "apiVersion": "v1",
  "temperature": 20,
  "humidity": 80,
  "city": "London"
}
```

# Potential Improvements

```
@Controller('/api')
class WeatherController {
  @Get('/weather')
  public get() {
    return {
      apiVersion: 'v1',
      temperature: 20,
      humidity: 80,
      city: 'London',
    }
  }
}
```

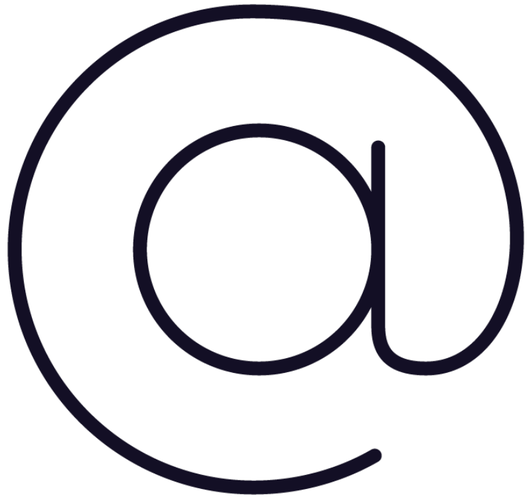
# Potential Improvements

```
@Controller('/api')
class WeatherController {
  @Get('/weather')
  public get() {
    return {
      apiVersion: 'v1',
      temperature: 20,
      humidity: 80,
      city: 'London',
    }
  }

  @Post('/submit-forecast')
  public post() {}
}
```

# Potential Improvements

```
@Controller('/api')
class WeatherController {
  @Get('/weather')
  public get(@Param city: string) {
    return {
      apiVersion: 'v1',
      temperature: 20,
      humidity: 80,
      city: 'London',
    }
  }

  @Post('/submit-forecast')
  public post(@Body requestBody: NewForecast) {}
}
```

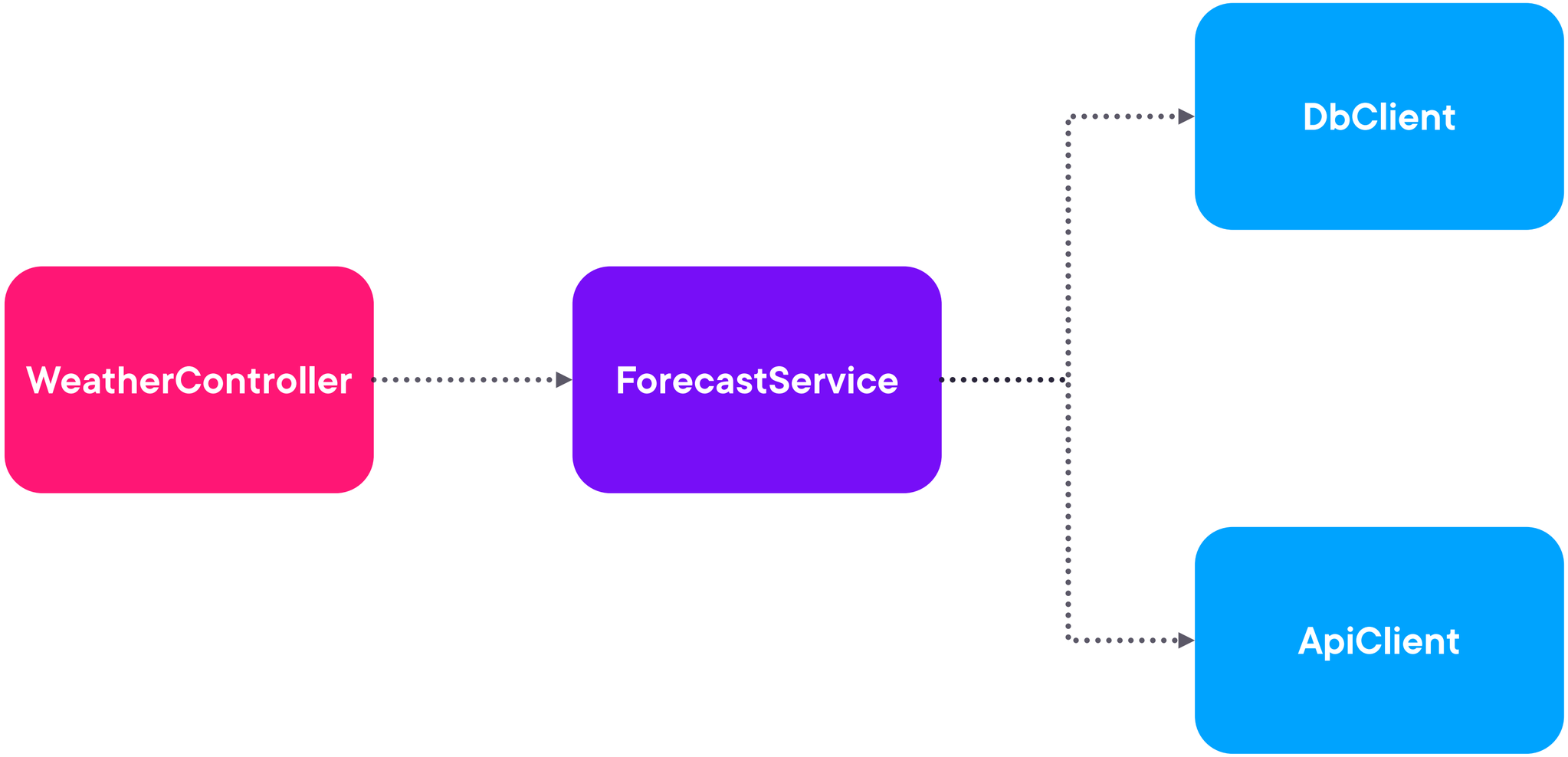
# Other Decorator Based Features



**Other Nest.js features**

- Inputs validation
- Authentication
- Caching
- Database schema definitions
- Tasks scheduling
- etc.

# Object Graph



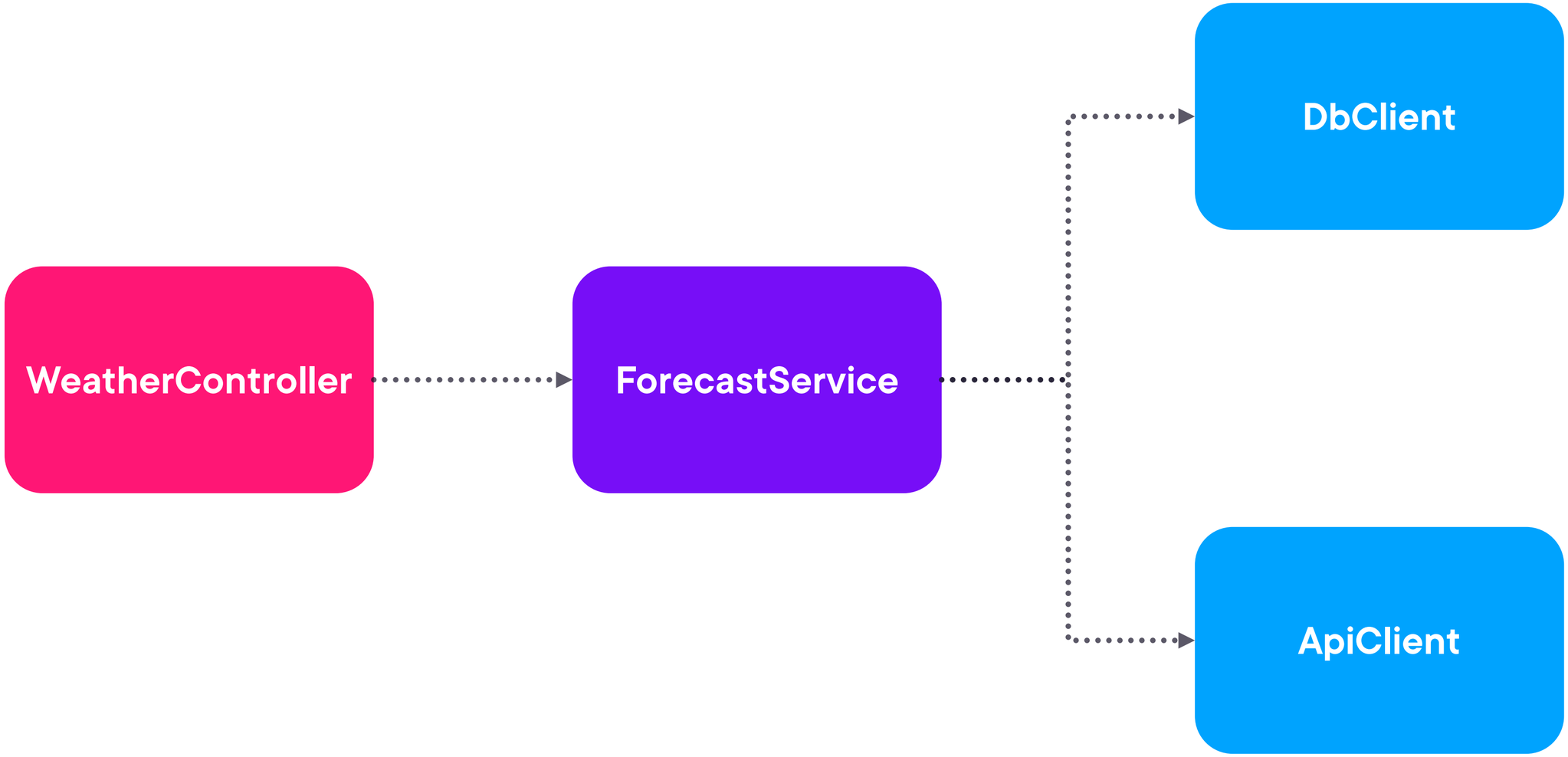
# Creating an Object Graph

```
const thirdPartyApiClient = new ThirdPartyApiClient(...);

const dbClient = new DbClient(...);

const forecastService = new ForecastService({
    dbClient: dbClient,
    thirdPartyApiClient: thirdPartyApiClient
});


const weatherController = new WeatherController({
    forecastService: forecastService
});
```

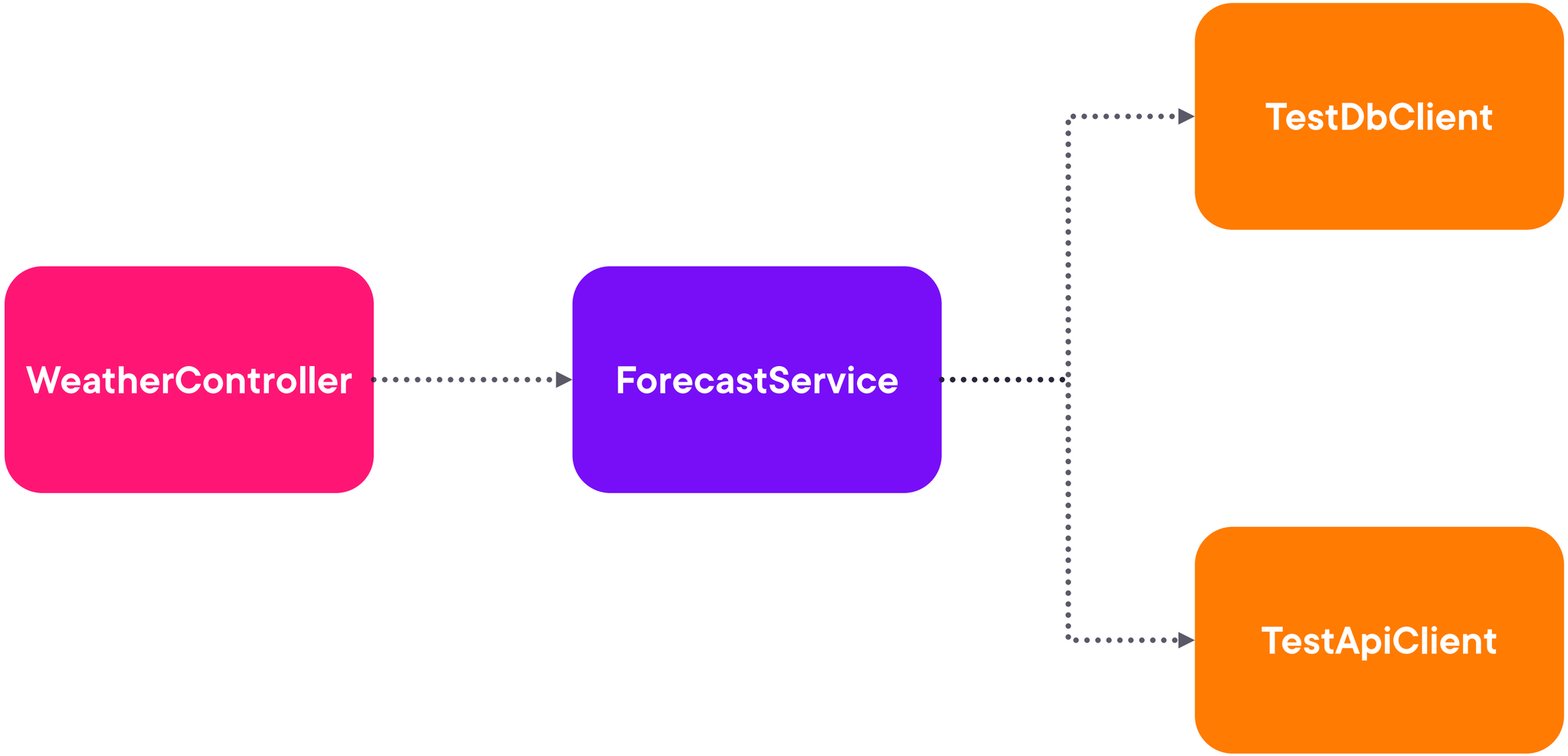
# Dependency Injection

```
@Controller('/weather')
class WeatherController {

  @Inject('WeatherService')
  private weatherService: WeatherService

  @Get
  public get() {
    return weatherStorage.getForecast()
  }
}

@Injectable('WeatherService')
class WeatherService {
  public getForecast() {
    return ...
  }
}
```

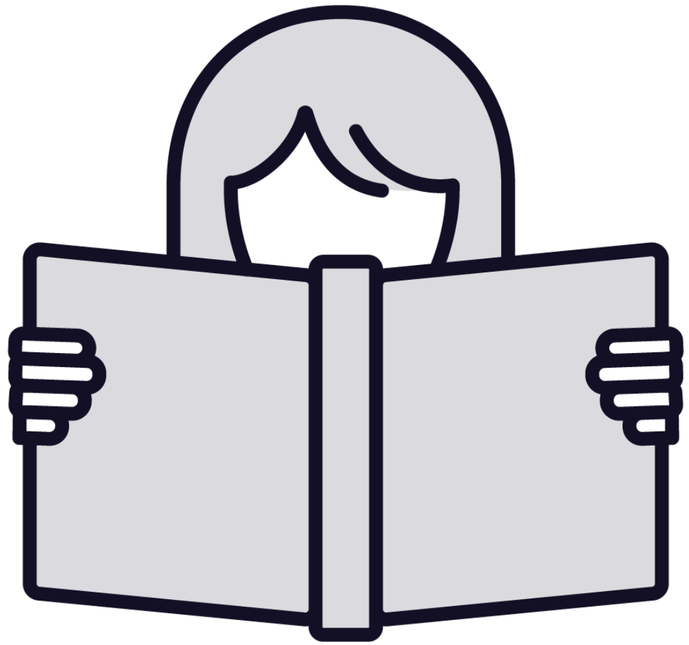
# Object Graph



# Test Objects



# To Read More



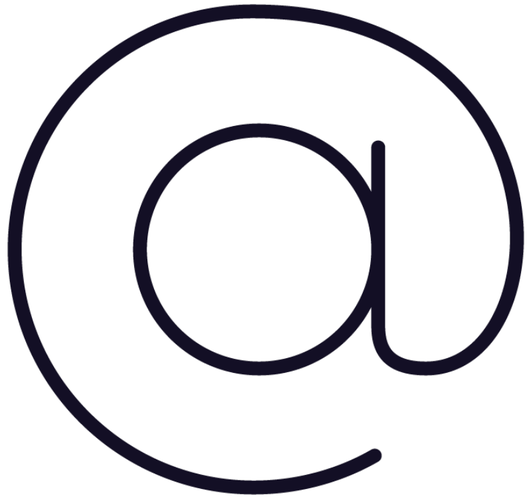
**Dependency Injection**

- https://docs.nestjs.com/providers

**How it works**

**How it is implemented in a real-world framework**

# Class Decorators



**Function**

- Receives a class
- [Optionally] returns a new class

**What can we do?**

- Record metadata
- Modify a class
- Execute code on instance creation

# Record Decorated Classes

```
Set<any> decoratedClasses = new HashSet<>()

function record(target: any, context: ClassDecoratorContext) {
  decocatedClasses.add(target)
}

@record
class MyClass {
  ...
}
```

# Class Decorator Context

```ts
interface ClassDecoratorContext<
    Class extends abstract new (...args: any) => any
> {
  /** The kind of element that was decorated. */
  readonly kind: "class"

  /** The name of the decorated class. */
  readonly name: string | undefined

  /**
   * Adds a callback to be invoked
   * after the class definition has been finalized.
   */
  addInitializer(initializer: (this: Class) => void): void

  readonly metadata: DecoratorMetadata
}
```

# Add New Field to a Class

```
function addNewField(target: any, context: any) {
  return class extends target {
    newField = 'newValue'
  }
}

@addNewField
class MyClass {
  ...
}

const myClass = new MyClass()
console.log((myClass as any).newField) // newValue
console.log(myClass instanceof MyClass)
```

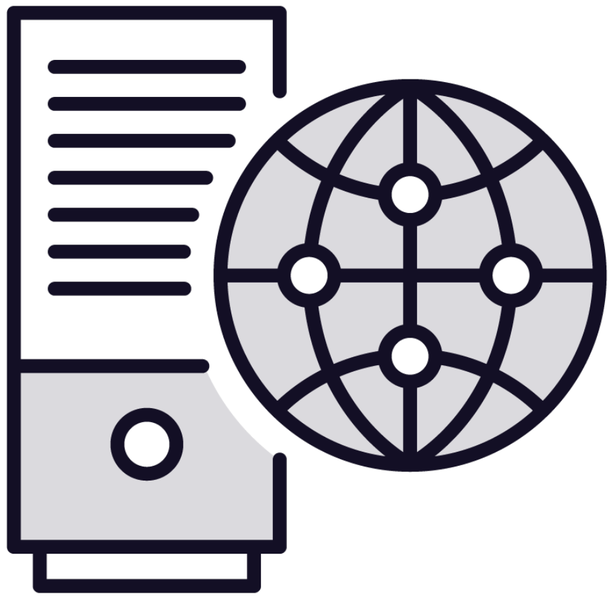
# Inspect Constructor Parameters

```
function inspectConstructor<T extends new (...args: any[]) => any>(
  target: T,
  context: ClassDecoratorContext<T>,
) {
  return class extends target {
    constructor(...args: any[]) {
      super(...args)
      console.log(`Args: ${args}`)
    }
  }
}

@inspectConstructor
class MyClass {
  constructor(name: string) {
    console.log('MyClass constructor')
  }
}
```

# Applying Decorators on Specific Classes

```
abstract class Controller {
  getCurrentUser(): User {
    ...
  }
}

function restrictDecorator(target: typeof Controller, context: any) {...}

@inspectConstructor
class WeatherController extends Controller {
  ...
}
```

# What Is Express.js?



**Popular Node.js web framework**

**Will build out web framework on top of it**

- Provide a more convenient API
- Call Express.js underneath

# Using Express.js

```
import express, { Request, Response } from 'express';

const app = express();

app.get('/user/current', (req: Request, res: Response) => {
  res.json({
    userId: 12345,
    name: 'John Doe',
    email: 'johndoe@example.com',
  });
});

app.listen(3000, () => {
  console.log('Server running on port 3000');
});
```

# Other Express.js Features

```
app.post('/user', (req: Request, res: Response) => {
  const { name, email } = req.body
  res.json({
    status: 'success',
  })
})
```

```
app.get('/user/:userId', (req: Request, res: Response) => {
  const userId = req.params.userId
  res.json({
    id: userId,
    ...
  })
})
```

# Demo

**Add Express.js to our project**

**Implement a simple API using it**

- Will rewrite it using our API later

# Controller Example

```
@Controller('/api')
class WeatherController {
  @Get('/weather')
  public get() {
    return {
      apiVersion: 'v1',
      temperature: 20,
      humidity: 80,
      city: 'London',
    }
  }
}
```

# How to Implement This?

**For @Controller**

- Store a map between controller class and route path

**For @Get**

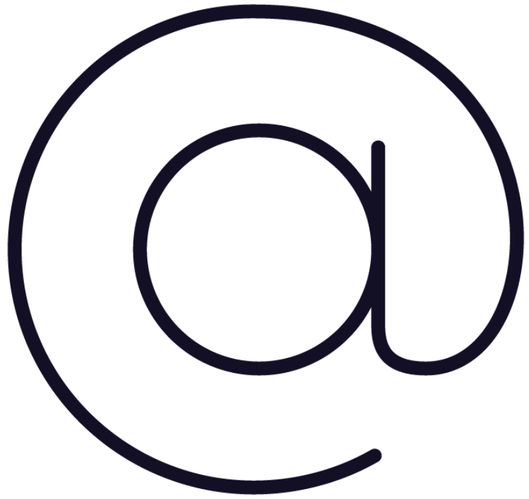
- Register an HTTP handler using Express.js
- Do it using the "`addInitializer`" method

# Demo

Implement @Controller and @Get decorators

Re-implement our web API using new decorators

# Properties Decorators



**Properties decorators**

**Accessors decorators**

- Decorators for getters and setters

**Auto-accessor decorators**

- What these are?
- How to implement decorators

# Property Decorator

```
type ClassPropertyDecorator = (
  target: undefined,
  context: {




  },
)
```

# Property Decorator

```
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
```

# Initialize Property

```
function notZero(_: any, context: any) {
  return function (initialValue: any) {
    if (initialValue === 0) {
        throw new Error('Initial value cannot be zero')
    }
    return initialValue
  }
}

class MyClass {
  @notZero
  myField: number = initValue()
}
```

# Getters and Setters

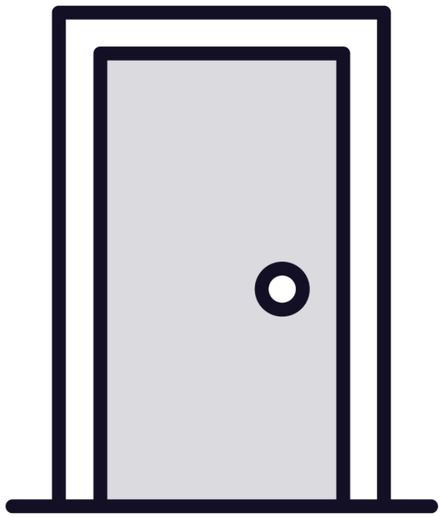
```
class User {
  private _email: string

  get email(): string {
    return this._email
  }

  set email(value: string) {
    this._email = value
  }
}
```

# Accessors Decorators

```ts
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
```

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

# Auto-Accessors



**Syntax for defining a field with accessors**

**Creates getters and setters automatically**

**Useful with decorators**

- Define initial value
- Can replace an auto-accessor

# Auto-Accessors

```
class User {
  private _email: string

  get email(): string {
    return this._email
  }

  set email(value: string) {
    this._email = value
  }
}
```

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

# Auto-Accessors Decorator

```
type ClassAutoAccessorDecorator = (
  target: {
    get: () => unknown
    set: (value: unknown) => void
  },









)
```

# Auto-Accessors Decorator

```
type ClassAutoAccessorDecorator = (
  target: {
    get: () => unknown
    set: (value: unknown) => void
  },
  context: {
    kind: 'accessor'
    name: string | symbol
    static: boolean
    private: boolean
    access: { get: () => unknown; set: (value: unknown) => void }
    addInitializer(initializer: () => void): void
  },
) => {
  get?: () => unknown
  set?: (value: unknown) => void
  init?: (initialValue: unknown) => unknown
} | void
```

# Auto-Accessor Decorator

```
function notNegative(target: any, context: any) {
  return {
    set(this: any, newValue: any) {
      if (newValue < 0) {
        throw new Error('Value cannot be 0')
      }

      target.set.call(this, newValue)
    },
  }
}

class User {
  @notNegative
  accessor age: number = 0
}
```

# Demo

Implement dependency injection decorators

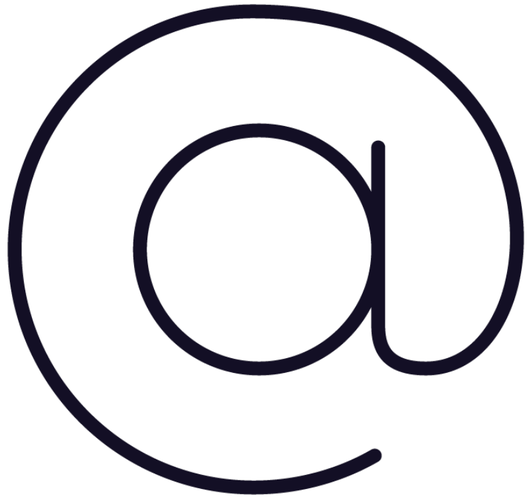
Created data source for a list of cities

Update our application to use new decorators

# Dependency Injection

```
@Controller('/weather')
class WeatherController {

  @Inject('WeatherDataStorage')
  private weatherStorage: WeatherDataStorage

  @Get
  public get() {
    return weatherStorage.getForecast()
  }
}

@Injectable('WeatherDataStorage')
class WeatherDataStorage {
  public getForecast() {
    return ...
  }
}
```

# How to Implement It



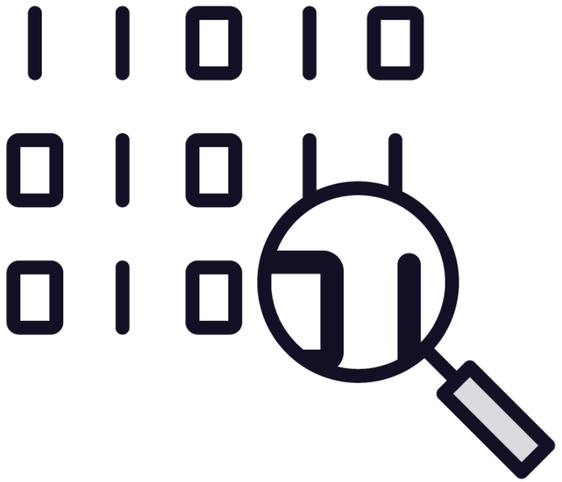
**Create a map between name of dependency and a class constructor**

- Write to this map in the @Injectable decorator

**@Inject will use this map**

- Create an instance using the constructor
- Set default value

# Decorators Metadata



**Introduced in TypeScript 5.2**

**Discuss what it is**

**Start with an example**

- See how detector metadata helps with it

# Data Validation

```
class User {
  @required
  username: string

  @required
  email: string

  location: string
}

function validate(object: any) {
  const requiredFields = getRequiredFields(object.constructor)

  for (const field of requiredFields) {
    if (!object[field]) {
      throw new Error(`${field} is required`)
    }
  }
}
```

# Validation without Metadata

```
type Class = { new(...args: any[]): any }

const requiredFields = new Map<Class, Set<string>>()

function required(target: any, context: any) {
  // How do we get the class here?
}
```

# Decorator Metadata

```
type Class = { new(...args: any[]): any }

const requiredFields = new Map<Class, Set<string>>()

function required(target: any, context: any) {
  context.addInitializer(function (this: any): void {
    // Update the required fields for this class
    // Will be called every time an object is created
  })
}
```

# Metadata Field

```
type ClassMethodDecorator = (
  value: Function,
  context: {
    kind: 'method'
    name: string | symbol
    static: boolean
    private: boolean
    access: { get: () => unknown }
    addInitializer(initializer: () => void): void
    metadata: Record<PropertyKey, unknown>
  },
) => Function | void
```

# Decorator Metadata

```
function decorator(target: any, context: any) {
  context.metadata[context.name] = true
}

class User {
    @decorator
    name: string = 'John'

    @decorator
    age: number = 42
}

console.log(User[Symbol.metadata])
```

# Decorator Metadata

```
function decorator(target: any, context: any) {
  context.metadata[context.name] = true
}

class User {
    @decorator
    name: string = 'John'

    @decorator
    age: number = 42
}

console.log(User[Symbol.metadata]) // { name: true, age: true }
```

# How to Use Metadata

```
function required(target: any, context: any) {
  const required = (context.metadata['requiredKeys'] ??= [])

  required.push(context.name)
}

function validate(object: any) {
  const metadata = object.constructor[Symbol.metadata]
  const requiredFields = metadata['requiredKeys']

  for (const field of requiredFields) {
    if (!object[field]) {
      throw new Error(`${field} is required`)
    }
  }
}
```

# Enabling Detector Metadata

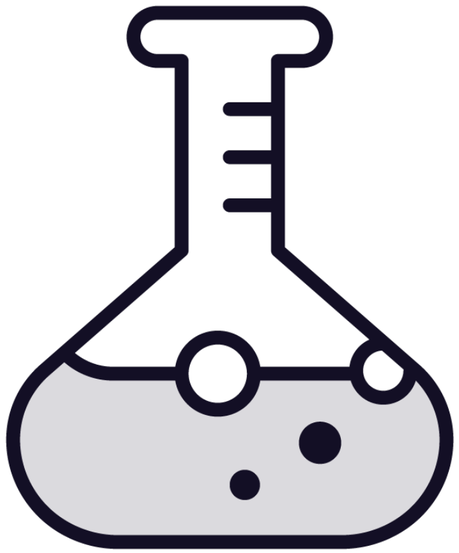
**tsconfig.json**

```
{
  "compilerOptions": {
    "target": "es2022",
    "lib": [
      "es2022",
      "esnext.decorators"
    ]
  }
}
```

**app.ts**

```
Symbol.metadata ??=
Symbol("Symbol.metadata");
```

# Experimental Decorators



**TypeScript 5.x implements a new decorators proposal**

**Pre-TypeScript 5.x had previous implementation of decorators**

- Have some advantages
- Still in use

```
// tsconfig.json

/* Enable experimental support for legacy experimental decorators. */
"experimentalDecorators": true,
```

## Experimental Decorators Before TypeScript 5.x

By default – decorators are disabled
With this configuration – uses old decorator's API

```
// tsconfig.json

/* Enable experimental support for legacy experimental decorators. */
"experimentalDecorators": true,
```

# Experimental Decorators in TypeScript 5.x

By default – uses new decorator's API
With this configuration – uses old decorator's API

# Experimental Decorators - Method Decorator

```
function logged(
  target: any,
  methodName: string,
  descriptor: PropertyDescriptor,
): {









}
```

# PropertyDescriptor

An object describing a class's property

```
interface PropertyDescriptor {
    configurable?: boolean;
    enumerable?: boolean;
    value?: any;
    writable?: boolean;
    get?(): any;
    set?(v: any): void;
}
```

# Experimental Decorators - Method Decorator

```
function logged(
  target: any,
  methodName: string,
  descriptor: PropertyDescriptor,
): {









}
```

# Experimental Decorators - Method Decorator

```
function logged(
  target: any,
  methodName: string,
  descriptor: PropertyDescriptor,
): void | PropertyDescriptor {
  const originalMethod = descriptor.value

  descriptor.value = function newMethod(this: any, ...args: any[]) {
    console.log(`@log - Running the ${methodName} method`)
    try {
      return originalMethod.apply(this, args)
    } finally {
      console.log(`@log - Method ${methodName} finished`)
    }
  }

  return descriptor
}
```

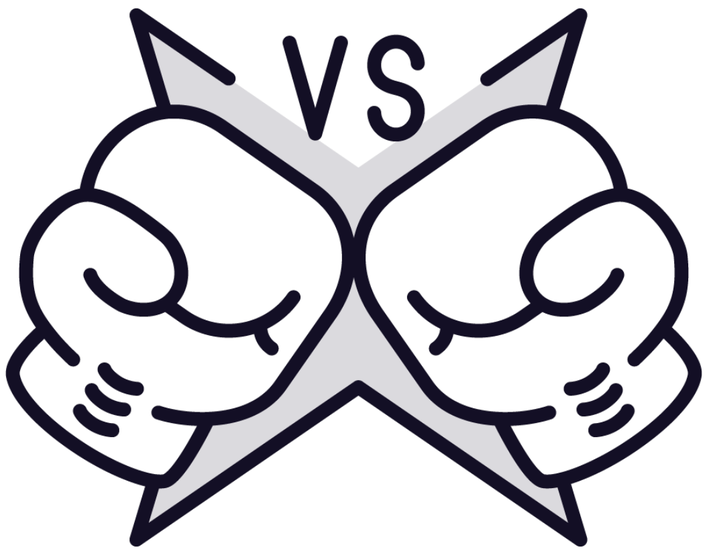
# Experimental Decorators - Property Decorator

```
function nonEnumerable(
  target: any,
  key: string,
  descriptor: PropertyDescriptor
) {

  descriptor.enumerable = false
}
```

# Experimental Decorators - Class Decorator

```
function classDecorator(target: any) {

  return class extends target {
    newField = 'newValue';
  }
}
```

# New vs. Experimental Decorators



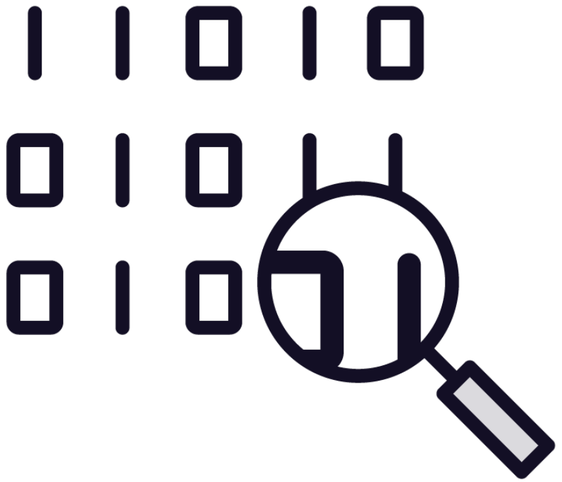

**Different APIs**

- Experimental decorators receive the "target" parameter
- Different return types

**Other differences**

- Reflect metadata
- Parameter decorators

# Decorators Metadata



**Automatically adding metadata on decorated classes/methods/etc.**

- Reflection – a program can examine its own structure
- Data about parameter types, return types, etc.

**Reflect Metadata API**

- Not exclusive to decorators

**How to enable it for TypeScript decorators**

- Only available for experimental decorators

# Reflect Metadata

```
import 'reflect-metadata'

class GitHubClient {
    async getUserInfo(username: string) {
        ...
    }
}

Reflect.defineMetadata(
    'thirdPartyClient',
    true,
    GitHubClient.prototype,
)
```

# Reflect Metadata

```
import 'reflect-metadata'

class GitHubClient {
    async getUserInfo(username: string) {
        ...
    }
}

Reflect.defineMetadata(
    'parameterTypes',
    [String],
    GitHubClient.prototype,
    'getUserInfo'
)
```

# Reading Metadata

```
Reflect.defineMetadata(
    'thirdPartyClient',
    true,
    GitHubClient.prototype,
)

Reflect.defineMetadata(
    'parameterTypes',
    [String],
    GitHubClient.prototype,
    'getUserInfo'
)


Reflect.getMetadata('thirdPartyClient', GitHubClient.prototype)
Reflect.getMetadata('parameterTypes', GitHubClient.prototype, 'getUserInfo')
```

```
/* Enable experimental support for legacy experimental decorators. */
"experimentalDecorators": true,
/* Emit design-type metadata for decorated declarations in source files. */
"emitDecoratorMetadata": true,
```

## Decorator's Metadata

Records metadata about decorators' targets
Method's parameter types, return types, etc.

# Detector Metadata

```
import 'reflect-metadata'

class WeatherService {
  @retry
  public get(city: string) {
    return ...
  }
}

// TypeScript compiler's output
__decorate([
    retry,
    __metadata("design:type", Function),
    __metadata("design:paramtypes", [String]),
    __metadata("design:returntype", void 0)
], WeatherService.prototype, "get", null);
```

# Detector Metadata

```
import 'reflect-metadata'

class WeatherService {
  @retry
  public get(city: string) {
    return ...
  }

  private getWeatherForecast(city: string) {...}
}

// TypeScript compiler's output
__decorate([
    retry,
    __metadata("design:type", Function),
    __metadata("design:paramtypes", [String]),
    __metadata("design:returntype", void 0)
], WeatherService.prototype, "get", null);
```

# Reading Detector Metadata

```
import 'reflect-metadata'

class WeatherService {
  @retry
  public get(city: string) {
    return ...
  }
}

const parameters = Reflect.getMetadata(
    'design:paramtypes',
    WeatherService.prototype,
    'get',
)
```

# Reading Detector Metadata

```
import 'reflect-metadata'

class WeatherService {
  @retry
  public get(city: string) {
    return ...
  }
}

const parameters = Reflect.getMetadata(
    'design:paramtypes',
    WeatherService.prototype,
    'get',
) // [String]
```

# Detector Metadata

| Record | Where? | Description |
| --- | --- | --- |

# Detector Metadata

| Record | Where? | Description |
| --- | --- | --- |
| **design:type** | methods/properties | Type of an object |

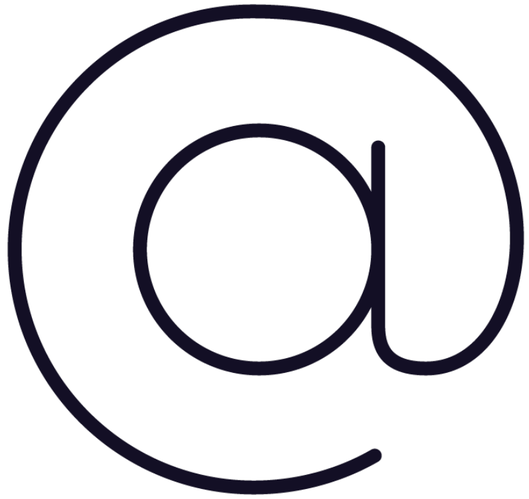
# Detector Metadata

| Record | Where? | Description |
| --- | --- | --- |
| **design:type** | methods/properties | Type of an object |
| **design:paramtypes** | methods/constructors | Input parameter types |

# Detector Metadata

| Record | Where? | Description |
| --- | --- | --- |
| **design:type** | methods/properties | Type of an object |
| **design:paramtypes** | methods/constructors | Input parameter types |
| **design:returntype** | methods | Return type |

# Decorators Metadata



**TypeScript does not provide metadata about methods/properties**

- Can't get a list of parameters programaticallly

**Useful when implementing decorators**

- Nest.js needs it to implement dependency injection

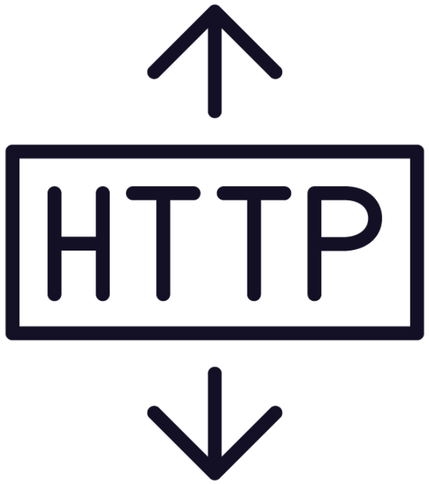
# Demo

Re-implement our framework using experimental decorators

Will see how to use experimental decorators API

Will allow us to add parameter decorators later

# Parameter Decorators




**Decorator for individual method parameters**

- Function called for each decorated parameter

**Support for query parameters in HTTP requests**

# Query Parameters

## Using parameter decorators

app.ts

```
@Controller('')
class WeatherController {
  @Get('/api')
  public get() {
    ...
  }
}
```

```
> wget \
app.io/api
```

# Query Parameters

## Using parameter decorators

app.ts

```
@Controller('')
class WeatherController {
  @Get('/api')
  public get() {
    ...
  }
}
```

```
> wget \
app.io/api?cityName=London
```

# Query Parameters

## Using parameter decorators

app.ts

```
@Controller('')
class WeatherController {
  @Get('/api')
  public get(
    @QueryParameter('cityName') city: string) {
    ...
  }
}
```

```
> wget \
app.io/api?cityName=London
```

# Query Parameters

## Using parameter decorators

app.ts

```
@Controller('')
class WeatherController {
  @Get('/api')
  public get(
    @QueryParameter('cityName') city: string) {
    return weatherClient.fetchWeather(city)
  }
}
```

```
> wget \
app.io/api?cityName=London
```

# Parameter Decorator

```
function QueryParameter(
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number,
) {
  const queryParameterMap = ...
  Reflect.defineMetadata(
    'additionalMetadata',
    queryParameterMap,
    target,
    propertyKey,
  )
}
```

# Implementing @QueryParameter

framework.ts

```
function QueryParameter(
  target: Object,
  propertyKey: string | symbol,
  parameterIndex: number,
) {
  const queryParameterMap = ...
  Reflect.defineMetadata(
    'additionalMetadata',
    queryParameterMap,
    target,
    propertyKey,
  )
}
```

app.ts

```
@Controller('/api')
class WeatherController {
  public get(
    @QueryParameter city: string) {
    ...
  }
}
```

# Implementing @QueryParameter

framework.ts

```
function QueryParameter(name: string)
{
  return function (
    target: Object,
    propertyKey: string | symbol,
    parameterIndex: number,
  ) {
    ...
  }
}
```

app.ts

```
@Controller('/api')
class WeatherController {
  public get(
    @QueryParameter city: string) {
    ...
  }
}
```

# Implementing @QueryParameter

framework.ts

```
function QueryParameter(name: string)
{
  return function (
    target: Object,
    propertyKey: string | symbol,
    parameterIndex: number,
  ) {
    ...
  }
}
```

app.ts

```
@Controller('/api')
class WeatherController {
  public get(
    @QueryParameter('city')
    city: string) {
    ...
  }
}
```

# Query Parameter Implementation

```
function QueryParameter(name: string) {
  return function (
    target: Object,
    propertyKey: string | symbol,
    parameterIndex: number,
  ) {
    const queryParameterMap: Map<number, string> =
      Reflect.getMetadata(
        'paramsMap',
        target,
        propertyKey) || new Map<number, string>()

    parametersMap.set(parameterIndex, queryParameterName)
    Reflect.defineMetadata('paramsMap', queryParameterMap, target, propertyKey)
  }
}
```

# Demo

Implement @QueryParameter

Add a query parameter to one of our controllers

# Summary - Decorators

```
function myDecorator(target, context) {
  return function (this, ...args) {
    return 'Hello world!'
  }
}
class GitHubClient {
  @myDecorator
  public async getRepos() {
    console.log('Getting GitHub repos...')
    ...
  }
}

console.log(new GitHubClient().getRepos()) // Hello world!
```

# Decorators APIs

**TypeScript 5 decorators**

New API
Enabled by default

**Experimental decorators**

Previous API version
Can be enabled

# Summary - Types of Decorators

Classes

Methods

Accessors

Properties

Auto-accessors

**Parameters**
(Experimental only)

# Summary - Applications of Method Decorators

Logging

Metrics/Tracing

Caching

Transactions

Callbacks

# Web Framework Using Decorators

```
@Controller('/api')
class WeatherController {
  @Get('/weather')
  public get() {
    return {
      apiVersion: 'v1',
      temperature: 20,
      humidity: 80,
      city: 'London',
    }
  }
}

startApp()
```

# Dependency Injection

```
@Controller('/weather')
class WeatherController {

  @Inject('WeatherDataStorage')
  private weatherStorage: WeatherDataStorage

  @Get
  public get() {
    return weatherStorage.getForecast()
  }
}

@Injectable('WeatherDataStorage')
class WeatherDataStorage {
  public getForecast() {
    return ...
  }
}
```

```
/* Enable experimental support for legacy experimental decorators. */
"experimentalDecorators": true,
/* Emit design-type metadata for decorated declarations in source files. */
"emitDecoratorMetadata": true,
```

## Experimental Decorators

Can be enabled in TypeScript 5
Can record decorator metadata