Decorators Revisited

Siddharth Ajmera **@SiddAjmera**

Content:

- Decorators Revisited
- Custom Decorators

Decorators Revisited

Angular offers 4 main types of Decorators:

- Class decorators, e.g. @Component and @NgModule
- Property decorators for properties inside classes, e.g. @Input and @Output
- Method decorators for methods inside classes, e.g. @HostListener
- Parameter decorators for parameters inside class constructors, e.g. @Inject

Class Decorators

- These are the top-level decorators that we use to express intent for classes.
- They allow us to tell Angular that a particular class is a component, or module, for example.
- And the decorator allows us to define this intent without having to actually put any code inside the class.
- @Component, @NgModule, @Directive, @Pipe, @Injectable decorators are the examples of class decorators.
- All we need to do is decorate it, and Angular will do the rest.

```
@Component({
  selector: 'example-component',
  template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
  constructor() {
    console.log('Hey I am a component!');
  }
}
```

```
@NgModule({
  imports: [],
  declarations: []
})
export class ExampleModule {
  constructor() {
    console.log('Hey I am a module!');
  }
}
```

Property Decorators

- Probably the second most common decorators.
- Allow us to decorate specific properties within our classes.
- We can simply put the @Input() decorator above the property.
- Angular's compiler will automatically create an input binding from the property name and link them.
- We'd then pass the input binding via a component property binding:
- The property decorator and "magic" happens within the ExampleComponent definition.
- In Angular there is a single property exampleProperty which is decorated, which is easier to change, maintain and track as our codebase grows.

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

@Component({
  selector: 'example-component',
  template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
  @Input()
  exampleProperty: string;
}
```

```
<example-component
  [exampleProperty]="exampleData">
</example-component>
```

Method Decorators

- Very similar to property decorators but are used for methods instead.
- Let's us decorate specific methods within our class with functionality.
- Eg: @HostListener that allows us to tell Angular that when an event on our host happens, we want the decorated method to be called with the event.

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

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

Parameter Decorators

- Used when injecting primitives into a constructor, where you need to manually tell Angular to inject a particular provider.
- Allow us to decorate parameters in our class constructors.
- Eg: @Inject lets us tell Angular what we want that parameter to be initiated with.
- Due to the metadata that TypeScript exposes for us we don't actually have to do this for our providers. We can just allow TypeScript and Angular to do the hard work for us by specifying the provider to be injected as the parameter type.

```
import { Component, Inject } from '@angular/core';
import { MyService } from './my-service';

@Component({
  selector: 'example-component',
  template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
  constructor(@Inject(MyService) myService) {
    console.log(myService); // MyService
  }
}
```

```
import { Component } from '@angular/core';
import { MyService } from './my-service';

@Component({
  selector: 'example-component',
  template: '<div>Woo a component!</div>'
})
export class ExampleComponent {
  constructor(myService: MyService) {
    console.log(myService); // MyService
  }
}
```

Custom Decorators

Decorators are functions that add something to the thing that is passed to them or are functions that returns the expression that will be called by the decorator at runtime. Since there are four things (class, parameter, method and property) that can be decorated; consequently there are four different function signatures for decorators:

- class: declare type ClassDecorator = <TFunction extends Function>(target: TFunction) => TFunction | void;
- property: declare type PropertyDecorator = (target: Object, propertyKey: string | symbol) => TFunction | void;
- method: declare type MethodDecorator = <T>(target: Object, propertyKey: string | symbol, descriptor: TypedPropertyDescriptor<T>) => TypedPropertyDescriptor<T> | void;
- parameter: declare type ParameterDecorator = (target: Object, propertyKey: string | symbol, parameterIndex: number) => void;

Custom Class Decorator

Use Case: We need a class decorator that will automagically unsubscribe from all the subscriptions that we have within our class.

```
function AutoUnsubscribe(constructor) {

  const original = constructor.prototype.ngOnDestroy;

  constructor.prototype.ngOnDestroy = function () {
    for ( let prop in this ) {
      const property = this[ prop ];
      if ( property && (typeof property.unsubscribe === "function") ) {
        property.unsubscribe();
      }
    }
    original && typeof original === "function" && original.apply(this, arguments);
  };

}
```

Custom Property Decorator

Use Case: We need to override a property within our class.

```
function Override(label: string) {
  return function (target: any, key: string) {
    Object.defineProperty(target, key, {
      configurable: false,
      set: () => target.key = label,
      get: () => label
    });
  }
}
```