@Directive, @Pipe & Custom Decorators

Siddharth Ajmera **@SiddAjmera**

Content:

- Directives
- Angular's Built-In Directives
- Custom Structural Directives
- Custom Attribute Directives
- Pipes
- Angular's Built-In Pipes
- Custom Pipes
- Decorators Revisited
- Custom Decorators

Directives

- Directives are a way of attaching behaviour to DOM elements
- Directives are decorated with the @Directive decorator
- Can be used in other components and directives
- Directives are registered in the Declarations array of an NgModule
- Directives are configured through the metadata passed to the @Directive decorator
- Directives can implement Lifecycle hooks to control their runtime behaviour

Types of Directives

- Components
  - Directives a template
  - Most common directive used throughout the app
- Structural Directives
  - Directives that change the DOM layout by adding/removing elements
  - Prefixed with an asterisk. Eg. *ngFor, *ngIf, *ngSwitch
- Attribute Directives
  - Directives that change the appearance or behaviour of an element
  - Used as attributes of elements. Eg. ngStyle, ngClass

Built-in Directives

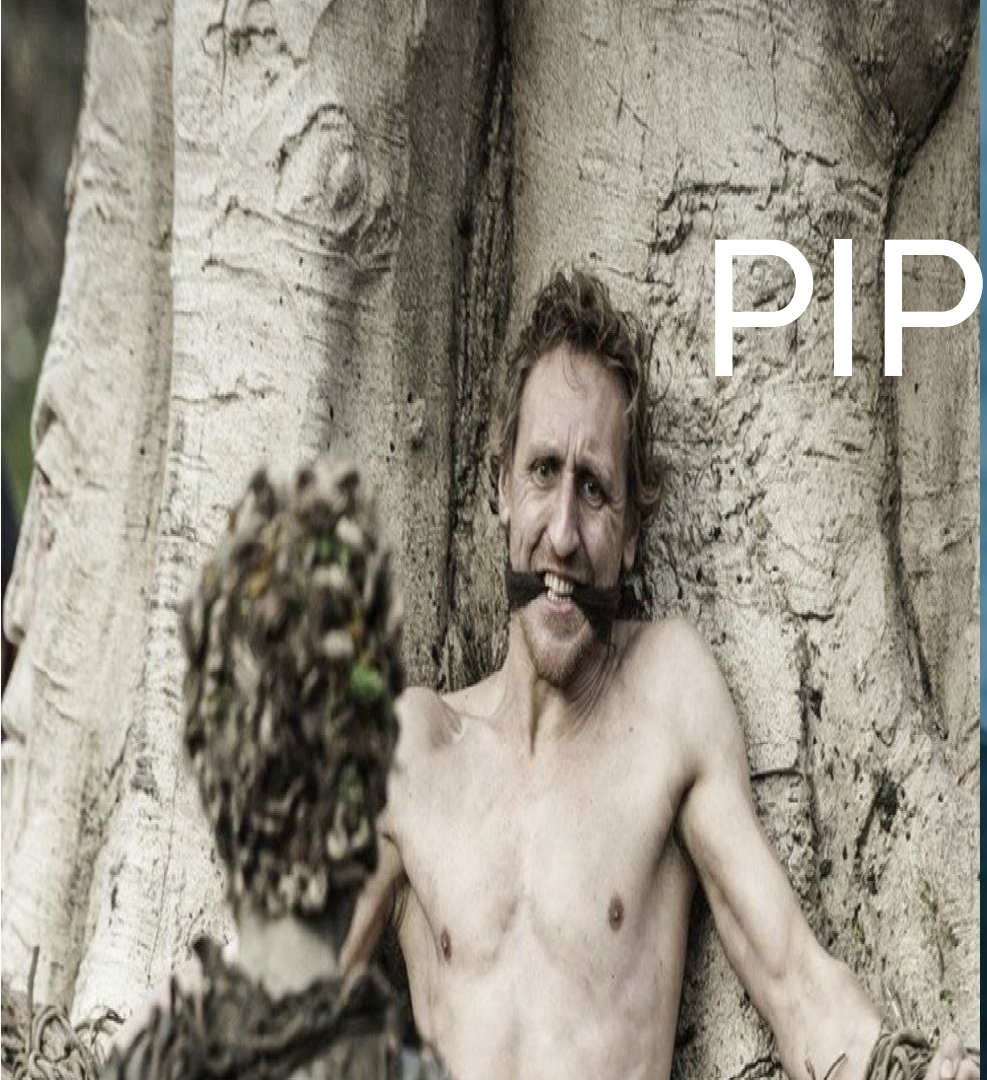
- Structural
  - NgFor
  - NgIf
  - NgSwitch

- Attribute
  - NgClass
  - NgStyle
  - NgNonBindable

Custom Directives

- Custom Directives are classes that are decorated with the @Directive decorator
- Contain the metadata 'selector' which is enclosed in [] to specify it as an attribute
- For attribute directives, we have access to the dom elements through elementRef, which can be updated. (Use renderer)
- We can access the properties of host element using @HostBinding, and register eventlisteners on the host element using @HostListerner
- We can pass data to the directive using @Input decorator
- In structural directives, we have access to the template as templateRef and to the view container as ViewContainerRef
- We can add the template to the container using vcRef.createEmbeddedView() and empty the container using vcRef.clear()

PIPES



Pipes

- Transforms some output into template.
- Think of it as a make-up room.
- Can handle both synchronous and asynchronous data.
- Don't change the value of the actual property itself. Just transforms the way it is presented on the UI.
- Since only responsible to transform the output, the logical place to use it, is the template.
- Several built in pipes provided as a part of Angular. Custom Pipes can also be built.
- Pipes are used by using the Pipe(|) symbol after the data to be transformed..
- Pipes can be chained with other pipes.
- Pipes can also be provided with arguments by using the colon (:) sign.

P AsyncPipe	P DatePipe	P I18nPluralPipe
P I18nSelectPipe	P JsonPipe	P LowerCasePipe
P CurrencyPipe	P DecimalPipe	P PercentPipe
P SlicePipe	P UpperCasePipe	P TitleCasePipe

Built in Pipes

UpperCasePipe

PercentPipe

AsyncPipe

DecimalPipe

DatePipe

TitleCasePipe

JsonPipe

CurrencyPipe

I18nPluralPipe

SlicePipe

I18nSelectPipe

LowerCasePipe

Custom Pipes

- Create a TypeScript Class with export keyword.
- Decorate it with the @Pipe decorator. Pass in the name property to its metadata.
- Implement the PipeTransform Interface on this class.
- Implement the transform method imposed due to the interface.
- Return the transformed data from the pipe.
- Add this pipe class to the declarations array of the module where you want to use it.
- OR simply use ng g p pipe-name. It will add the bare-bones of a pipe to your project and will also update your root module.

```
PS C:\Development\Angular\Week2Assignment> ng g p shortenPipe
installing pipe
  create src\app\shorten-pipe.pipe.spec.ts
  create src\app\shorten-pipe.pipe.ts
  update src\app\app.module.ts
PS C:\Development\Angular\Week2Assignment>
```

- You can also add arguments to your pipe by adding them to the transform function as parameters.
- By default the pipes are pure in nature. To change it and update view as the data changes, make them impure by adding the pure property to the metadata and setting it to false.
- Use AsyncPipe to handle promises or observables.

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
    });
  }
}
```