Custom Pipes

Learning Objectives

- Know how to use the `@Pipe` decorator to create pipes.
- Know how to pass in parameters to custom pipes.

Pipe decorator

One pipe I really find useful when building web applications is a *default* pipe, which I use for things like avatar images.

I use this pipe in an image field, like so:

```
<img [src]="imageUrl | default:'<default-image-url>'"/>
```

The pipe is called `default` and we pass to it a default image to use if the `imageUrl` variable is blank.

To create a pipe we use the `@Pipe` decorator and annotate a class like so:

```
import { Pipe } from '@angular/core';
.
.
.
@Pipe({
  name:"default"
})
class DefaultPipe { }
```

The name parameter for the `Pipe` decorator is how the pipe will be called in templates.

Transform function

The actual logic for the pipe is put in a function called `transform` on the class, like so:

```
class DefaultPipe {
  transform(value: string, fallback: string): string {
    let image = "";
    if (value) {
      image = value;
    } else {
      image = fallback;
    }
     return image;
  }
}
```

The first argument to the transform function is the *value* that is passed *into* the pipe, i.e. the thing that goes *before* the `|` in the expression.

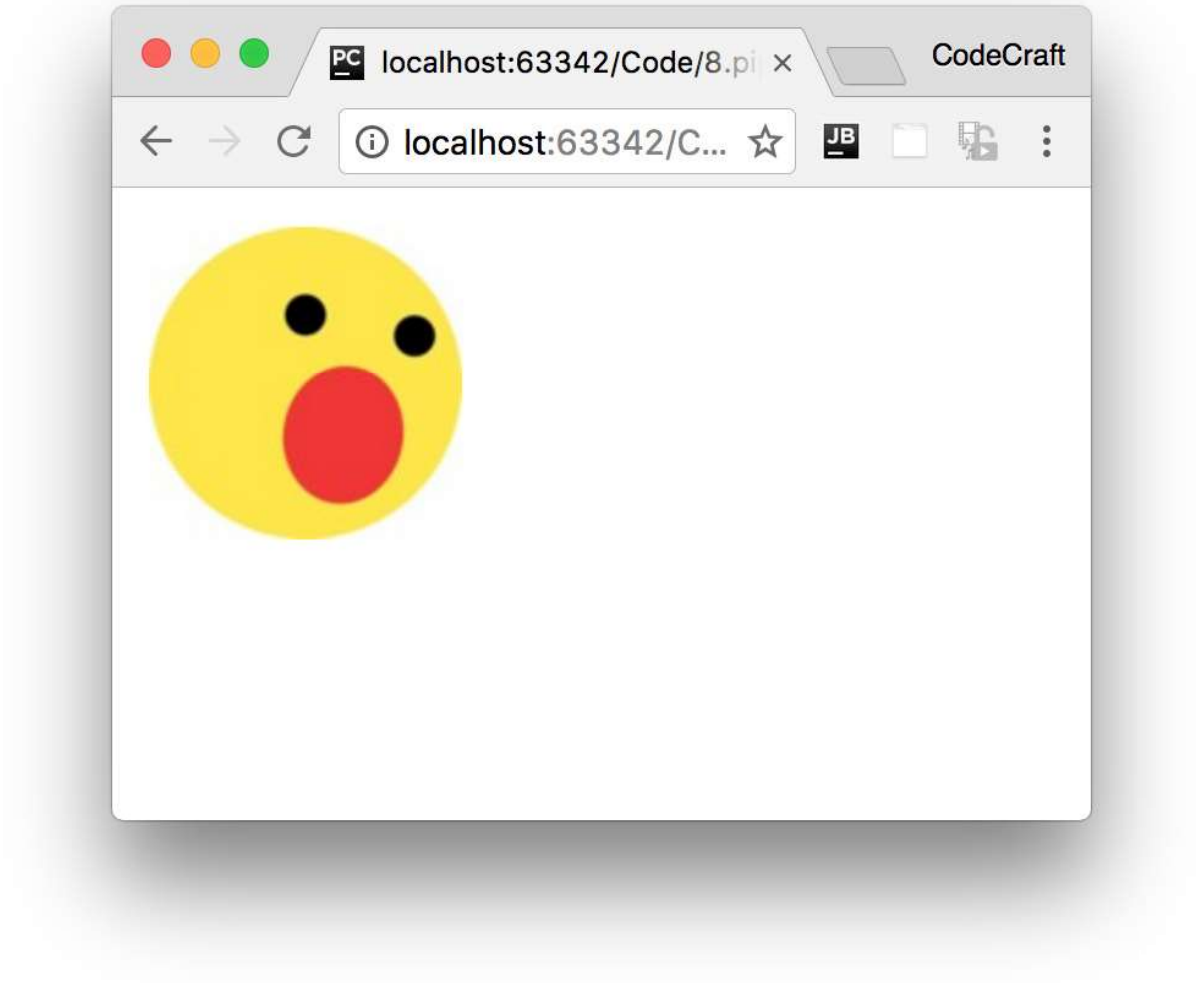
The second parameter to the transform function is the first param we pass into our pipe, i.e. the thing that goes after the `:` in the expression.

Specifically with this example:

```
@Component({
  selector: 'app',
  template: `
  <img [src]="imageUrl |
default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg'"/>
 `
})
class AppComponent {
  imageUrl: string = "";
}
```

- `value` gets passed `imageUrl` which is blank.
- `fallback` gets passed 'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg'

When we run the above, since `imageUrl` is blank the `default` pipe uses the default image which is passed in, like so:



Multiple parameters

Finally we want to support an *optional* third param to our pipe called `forceHttps`, if the image selected doesn't use *https* the pipe will convert the url to one that does use *https*.

To support additional parameters in pipes we just add more parameters to our *transform* function.

Because this one is optional and we are using TypeScript we can define the new param and also give it a default value of `false`.

```
class DefaultPipe {
  transform(value: string, fallback: string, forceHttps: boolean = false): string {
    let image = "";
    if (value) {
      image = value;
    } else {
      image = fallback;
    }
    if (forceHttps) {
      if (image.indexOf("https") == -1) {
        image = image.replace("http", "https");
      }
    }
    return image;
  }
}
```

And to use this optional param we just extend the pipe syntax in our template with another `:`, like so:-

```
<img [src]="imageUrl |
default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg':true"/> ①
```

① Notice the last `:true` at the end of the pipe expression.

Now we force the image url to use the https protocol.

Summary

Creating a pipe is very simple in Angular. We just decorate a class with the `@Pipe` decorator, provide a name and a transform function and we are done.

Listing

script.ts

```
import {NgModule, Component, Pipe} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {Observable} from 'rxjs/Rx';


@Pipe({
  name: "default"
})
class DefaultPipe {
  transform(value: string, fallback: string, forceHttps: boolean = false): string {
    let image = "";
```

```
    if (value) {
      image = value;
    } else {
      image = fallback;
    }

    if (forceHttps) {
      if (image.indexOf("https") == -1) {
        image = image.replace("http", "https");
      }
    }

    return image;
  }
}

@Component({
  selector: 'app',
  template: `
  <img [src]="imageUrl |
default:'http://s3.amazonaws.com/uifaces/faces/twitter/sillyleo/128.jpg':true"/>
 `
})
class AppComponent {
  imageUrl: string = "";
}


@NgModule({
  imports: [BrowserModule],
  declarations: [AppComponent,
    DefaultPipe
  ],
  bootstrap: [AppComponent],
})
class AppModule {

}

platformBrowserDynamic().bootstrapModule(AppModule);
```

Wrapping Up

Pipes are a way of having a different *visual representation* for the same piece of data without storing unnecessary intermediate data on the component.

We could solve the same problems without using pipes. We can transform the data ourselves on the component into different property, in all the different permutations they are required in all the views they are consumed in.

But that's wasteful and a fertile feeding ground for bugs.

Instead we use pipes which are used in templates and transform the data passed to it on demand in the format the consumer of the data wants without needing to store any intermediate values on our component.