PIPES

Pipes transform displayed values within a template.

Every application starts out with what seems like a simple task: get data, transform them, and show them to users.

Getting data could be as simple as creating a local variable or as complex as streaming data over a Websocket.

Once data arrive, we could push their raw `toString` values directly to screen. That rarely makes for a good user experience. Almost everyone prefers a simple birthday date (`April 15, 1988`) to the original raw string format (`Fri Apr 15 1988 00:00:00 GMT-0700 (Pacific Daylight Time)`).

Clearly some values benefit from a bit of massage. We soon discover that we desire many of the same transformations repeatedly, both within and across many applications. We almost think of them as styles. In fact, we'd like to apply them in our HTML templates as we do styles.

Welcome, Angular pipes, the simple display-value transformations that we can declare in our HTML!

Live Example.

Using Pipes

A pipe takes in data as input and transforms it to a desired output. We'll illustrate by transforming a component's birthday property into a human-friendly date:

app/hero-birthday1.component.ts

```
1. import {Component} from 'angular2/core'
2.
3. @Component({
4.   selector: 'hero-birthday',
5.   template: `<p>The hero's birthday is {{ birthday | date }}</p>`
6. })
7. export class HeroBirthday {
8.   birthday = new Date(1988,3,15); // April 15, 1988
9. }
```

Focus on the component's template.

```
<p>The hero's birthday is {{ birthday | date }}</p>
```

Inside the interpolation expression we flow the component's `birthday` value through the pipe operator (|) to the Date pipe function on the right. All pipes work this way.

Built-in pipes

Angular comes with a stock set of pipes such as `DatePipe`, `UpperCasePipe`, `LowerCasePipe`, `CurrencyPipe`, and `PercentPipe`. They are all immediately available for use in any template.

> Learn more about these and many other built-in pipes in the the API Reference; filter for entries that include the word "pipe".

Parameterizing a Pipe

A pipe may accept any number of optional parameters to fine-tune its output.

We add parameters to a pipe by following the pipe name with a colon (:) and then the parameter value (e.g., `currency:'EUR'`). If our pipe accepts multiple parameters, we separate the values with colons (e.g. `slice:1:5`)

We'll modify our birthday template to give the date pipe a format parameter. After formatting the hero's April 15th birthday should display as **`04/15/88`**.

```
<p>The hero's birthday is {{ birthday | date:"MM/dd/yy" }} </p>
```

The parameter value can be any valid template expression such as a string literal or a component property. In other words, we can control the format through a binding the same way we control the birthday value through a binding.

Let's write a second component that *binds* the pipe's format parameter to the component's `format` property. Here's the template for that component:

app/hero-birthday2.component.ts (template)

```
template: `
  <p>The hero's birthday is {{ birthday | date:format }}</p>
  <button (click)="toggleFormat()">Toggle Format</button>
`
```

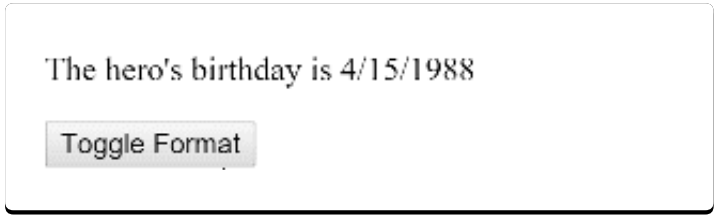
We also added a button to the template and bound its click event to the component's `toggleFormat` method. That method toggles the component's `format` property between a short form ('shortDate') and a longer form ('fullDate').

app/hero-birthday2.component.ts (class)

```
1. export class HeroBirthday {
2.   birthday = new Date(1988,3,15); // April 15, 1988
3.
4.   toggle = true; // start with true == shortDate
5.
6.   get format() { return this.toggle ? 'shortDate' : 'fullDate'}
7.
8.   toggleFormat() { this.toggle = !this.toggle; }
```

```
9. }
```

As we click the button, the displayed date alternates between "**`04/15/1988`**" and "**`Friday, April 15, 1988`**".



> Learn more about the `DatePipes` format options in the API Docs.

Chaining pipes

We can chain pipes together in potentially useful combinations. In the following example, we chain the birthday to the `DatePipe` and on to the `UpperCasePipe` so we can display the birthday in uppercase. The following birthday displays as **`APR 15, 1988`**

```
<p>
  The chained hero's birthday is
  {{ birthday | date | uppercase}}
</p>
```

If we pass a parameter to a filter, we have to add parentheses to help the template compiler with the evaluation order. The

following example displays **`FRIDAY, APRIL 15, 1988`**

```
<p>
  The chained hero's birthday is
  {{  birthday | date:'fullDate' | uppercase}}
</p>
```

We can add parentheses to alter the evaluation order or to provide extra clarity:

```
<p>
  The chained hero's birthday is
  {{ ( birthday | date:'fullDate' ) | uppercase}}
</p>
```

Custom Pipes

We can write our own custom pipes.

Here's a custom pipe named `ExponentialStrengthPipe` that can boost a hero's powers:

app/exponential-strength.pipe.ts

```
1. import {Pipe, PipeTransform} from 'angular2/core';
```

```
/*
 * Raise the value exponentially
 * Takes an exponent argument that defaults to 1.
 * Usage:
 *   value | exponentialStrength:exponent
 * Example:
 *   {{ 2 |  exponentialStrength:10}}
 *   formats to: 1024
*/
@Pipe({name: 'exponentialStrength'})
export class ExponentialStrengthPipe implements PipeTransform {

  transform(value:number, args:string[]) : any {
    return Math.pow(value, parseInt(args[0] || '1', 10));
  }
}
```

This pipe definition reveals several key points

- A pipe is a class decorated with pipe metadata.
- The pipe class implements the `PipeTransform` interface's `transform` method that takes an input value and an optional array of parameter strings and returns the transformed value.
- There will be one item in the parameter array for each parameter passed to the pipe
- We tell Angular that this is a pipe by applying the `@Pipe` decorator which we import from the core Angular library.
- The `@Pipe` decorator takes an object with a name property whose value is the pipe name that we'll use within a

template expression. It must be a valid JavaScript identifier. Our pipe's name is `exponentialStrength`.

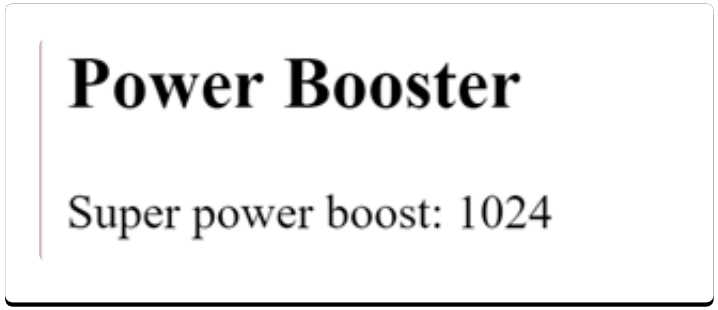
The *PipeTransform* Interface

The `transform` method is essential to a pipe. The `PipeTransform` interface defines that method and guides both tooling and the compiler. It is optional; Angular looks for and executes the `transform` method regardless.

Now we need a component to demonstrate our pipe.

app/power-booster.component.ts

```
1. import {Component} from 'angular2/core';
2. import {ExponentialStrengthPipe} from './exponential-strength.pipe';
3.
4. @Component({
5.   selector: 'power-booster',
6.   template: `
7.     <h2>Power Booster</h2>
8.     <p>
9.       Super power boost: {{2 | exponentialStrength: 10}}
10.    </p>
11.  `,
12.  pipes: [ExponentialStrengthPipe]
13. })
14. export class PowerBooster { }
```



Two things to note:

1. We use our custom pipe the same way we use the built-in pipes.
2. We must list our pipe in the @Component decorator's `pipes` array.

REMEMBER THE PIPES ARRAY!

Angular reports an error if we neglect to list our custom pipe. We didn't list the `DatePipe` in our previous example because all Angular built-in pipes are pre-registered. Custom pipes must be registered manually.

If we try the live code example, we can probe its behavior by changing the value and the optional exponent in the template.

Power Boost Calculator (extra-credit)

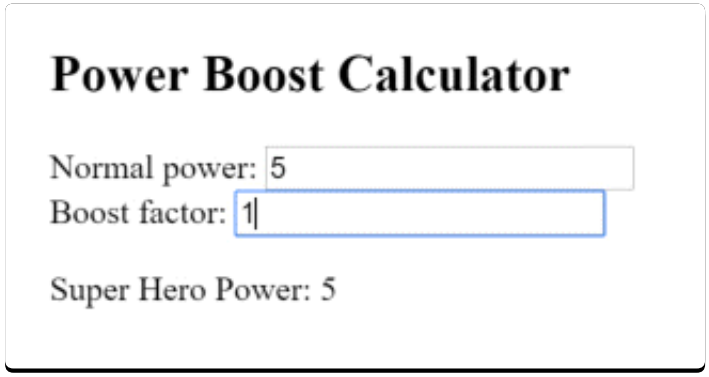
It's not much fun updating the template to test our custom pipe. We could upgrade the example to a "Power Boost Calculator" that combines our pipe and two-way data binding with `ngModel`.

/app/power-boost-calculator.component.ts

```
1. import {Component} from 'angular2/core';
```

```
import {ExponentialStrengthPipe} from './exponential-strength.pipe';

@Component({
  selector: 'power-boost-calculator',
  template: `
    <h2>Power Boost Calculator</h2>
    <div>Normal power: <input [(ngModel)]="power"></div>
    <div>Boost factor: <input [(ngModel)]="factor"></div>
    <p>
      Super Hero Power: {{power | exponentialStrength: factor}}
    </p>
  `,
  pipes: [ExponentialStrengthPipe]
})
export class PowerBoostCalculator {
  power = 5;
  factor = 1;
}
```



Stateful Pipes

There are two categories of pipes, stateless and stateful.

Stateless pipes are pure functions that flow input data through without remembering anything or causing detectable side-effects.

Most pipes are stateless. The `DatePipe` in our first example is a stateless pipe. So is our custom `ExponentialStrengthPipe`.

Stateful pipes are conceptually similar to classes in object-oriented programming. They can manage the data they transform. A pipe that creates an HTTP request, stores the response and displays the output, is a stateful pipe. Pipes that retrieve or request data should be used cautiously, since working with network data tends to introduce error conditions that are better handled in JavaScript than in a template. We can mitigate this risk by creating a custom pipe for a particular backend and bake-in the essential error-handling.

The stateful `AsyncPipe`

The Angular Async pipe is a remarkable example of a stateful pipe. The Async pipe can receive a Promise or Observable as input and subscribe to the input automatically, eventually returning the emitted value(s).

It is stateful because the pipe maintains a subscription to the input and its returned values depend on that subscription.

In the next example, we bind a simple promise to a view with the async pipe.

app/hero-async-message.component.ts

```
import {Component} from 'angular2/core';

// Initial view: "Message: "
// After 500ms: Message: You are my Hero!"

@Component({
  selector: 'hero-message',
  template: 'Message: {{delayedMessage | async}}',
})
export class HeroAsyncMessageComponent {
  delayedMessage:Promise<string> = new Promise((resolve, reject) => {
    setTimeout(() => resolve('You are my Hero!'), 500);
  });
}
```

The Async pipe saves boilerplate in the component code. The component doesn't have to subscribe to the async data source, it doesn't extract the resolved values and expose them for binding, and (in the case of Observable stream sources like `EventEmitter`) the component doesn't have to unsubscribe when it is destroyed (a potent source of memory leaks).

Implementing a Stateful Pipe

Pipes are stateless by default. We must declare a pipe to be stateful by setting the `pure` property of the `@Pipe` decorator to `false`. This setting tells Angular's change detection system to check the output of this pipe each cycle, whether its input has changed or not.

Here's how we'll decorate our new stateful `FetchJsonPipe` that makes an HTTP `fetch` request and (eventually) displays the data in the server's response:

app/fetch-json.pipe.ts (metadata)

```
@Pipe({
  name: 'fetch',
  pure: false
})
```

Immediately below we have the finished pipe. Its input value is an url to an endpoint that returns a JSON file. The pipe makes a one-time async request to the server and eventually receives the JSON response.

app/fetch-json.pipe.ts

```
import {Pipe, PipeTransform} from 'angular2/core';

@Pipe({
  name: 'fetch',
  pure: false
})
export class FetchJsonPipe  implements PipeTransform{
  private fetchedValue:any;
  private fetchPromise:Promise<any>;

  transform(value:string, args:string[]):any {
    if (!this.fetchPromise) {
      this.fetchPromise = window.fetch(value)
        .then((result:any) => result.json())
        .then((json:any)   => this.fetchedValue = json);
    }

```

```
    return this.fetchedValue;
  }
}
```

Next we demonstrate this pipe in a test component whose template defines two bindings

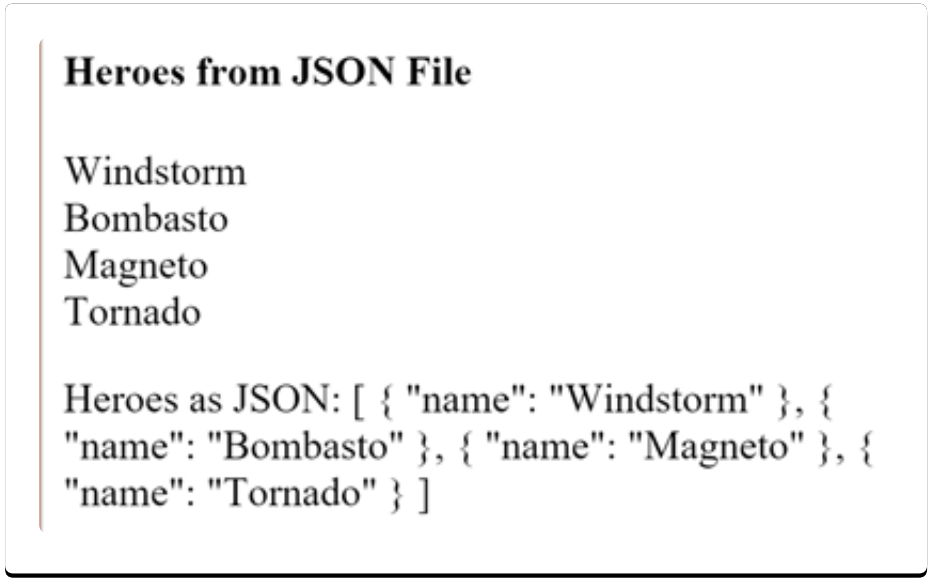
app/hero-list.component.ts (template)

```
template: `
  <h4>Heroes from JSON File</h4>

  <div *ngFor="#hero of ('heroes.json' | fetch) ">
    {{hero.name}}
  </div>

  <p>Heroes as JSON:
  {{'heroes.json' | fetch | json}}
  </p>
`,
```

The component renders like this:



The first binding is straight forward. An `ngFor` repeater displays the hero names fetched from a json source file. We're piping the literal file name, "heroes.json", through to the custom `fetch` pipe.

JsonPipe

The second binding uses more pipe chaining. We take the same fetched results and display the raw hero data in JSON format by piping to the built-in `JsonPipe`.

DEBUGGING WITH THE JSON PIPE

The JsonPipe is an easy way to diagnosis a mysteriously failing data binding.

Here's the complete component implementation:

app/hero-list.component.ts

```
import {Component} from 'angular2/core';
import {FetchJsonPipe} from './fetch-json.pipe';

@Component({
  selector: 'hero-list',
  template: `
    <h4>Heroes from JSON File</h4>

    <div *ngFor="#hero of ('heroes.json' | fetch) ">
      {{hero.name}}
    </div>

    <p>Heroes as JSON:
    {{'heroes.json' | fetch | json}}
    </p>
  `,
  pipes: [FetchJsonPipe]
})
export class HeroListComponent {
  /* I've got nothing to do ;-) */
}
```

Next Steps

Pipes are a great way to encapsulate and share common display-value transformations. We use them like styles, dropping them into our templates expressions to enrich the appeal and usability of our views.

Explore Angular's inventory of built-in pipes in the API Reference. Try writing a custom pipe and perhaps contributing it to the community.

Next Step

Routing & Navigation