Transforming Data Using Pipes

Use pipes to transform strings, currency amounts, dates, and other data for display. Pipes are simple functions to use in template expressions to accept an input value and return a transformed value. Pipes are useful because you can use them throughout your application, while only declaring each pipe once. For example, you would use a pipe to show a date as **April 15, 1988** rather than the raw string format.

For the sample application used in this topic, see the .

Angular provides built-in pipes for typical data transformations, including transformations for internationalization (i18n), which use locale information to format data. The following are commonly used built-in pipes for data formatting:

- `DatePipe`: Formats a date value according to locale rules.
- `UpperCasePipe`: Transforms text to all upper case.
- `LowerCasePipe`: Transforms text to all lower case.
- `CurrencyPipe`: Transforms a number to a currency string, formatted according to locale rules.
- `DecimalPipe`: Transforms a number into a string with a decimal point, formatted according to locale rules.
- `PercentPipe`: Transforms a number to a percentage string, formatted according to locale rules.

- For a complete list of built-in pipes, see the pipes API documentation.
- To learn more about using pipes for internationalization (i18n) efforts, see formatting data based on locale.

Create pipes to encapsulate custom transformations and use your custom pipes in template expressions.

Prerequisites

To use pipes you should have a basic understanding of the following:

- Typescript and HTML5 programming
- Templates in HTML with CSS styles
- Components

Using a pipe in a template

To apply a pipe, use the pipe operator (`|`) within a template expression as shown in the following code example, along with the *name* of the pipe, which is `date` for the built-in `DatePipe`. The tabs in the example show the following:

- `app.component.html` uses `date` in a separate template to display a birthday.
- `hero-birthday1.component.ts` uses the same pipe as part of an in-line template in a component that also sets the birthday value.

The component's `birthday` value flows through the pipe operator, `|` to the `date` function.

{@a parameterizing-a-pipe}

Transforming data with parameters and chained pipes

Use optional parameters to fine-tune a pipe's output. For example, use the `CurrencyPipe` with a country code such as EUR as a parameter. The template expression `{{ amount | currency:'EUR' }}` transforms the `amount` to currency in euros. Follow the pipe name (`currency`) with a colon (`:`) and the parameter value (`'EUR'`).

If the pipe accepts multiple parameters, separate the values with colons. For example, `{{ amount | currency:'EUR':'Euros '}}` adds the second parameter, the string literal `'Euros '`, to the output string. Use any valid template expression as a parameter, such as a string literal or a component property.

Some pipes require at least one parameter and allow more optional parameters, such as `SlicePipe`. For example, `{{ slice:1:5 }}` creates a new array or string containing a subset of the elements starting with element `1` and ending with element `5`.

Example: Formatting a date

The tabs in the following example demonstrates toggling between two different formats (`'shortDate'` and `'fullDate'`):

- The `app.component.html` template uses a format parameter for the `DatePipe` (named `date`) to show the date as **04/15/88**.
- The `hero-birthday2.component.ts` component binds the pipe's format parameter to the component's `format` property in the `template` section, and adds a button for a click event bound to the component's `toggleFormat()` method.
- The `hero-birthday2.component.ts` component's `toggleFormat()` method toggles the component's `format` property between a short form (`'shortDate'`) and a longer form (`'fullDate'`).

Clicking the **Toggle Format** button alternates the date format between **04/15/1988** and **Friday, April 15, 1988**.

For `date` pipe format options, see DatePipe.

Example: Applying two formats by chaining pipes

Chain pipes so that the output of one pipe becomes the input to the next.

In the following example, chained pipes first apply a format to a date value, then convert the formatted date to uppercase characters. The first tab for the `src/app/app.component.html` template chains `DatePipe` and

`UpperCasePipe` to display the birthday as **APR 15, 1988**. The second tab for the `src/app/app.component.html` template passes the `fullDate` parameter to `date` before chaining to `uppercase`, which produces **FRIDAY, APRIL 15, 1988**.

{@a Custom-pipes}

Creating pipes for custom data transformations

Create custom pipes to encapsulate transformations that are not provided with the built-in pipes. Then, use your custom pipe in template expressions, the same way you use built-in pipes—to transform input values to output values for display.

Marking a class as a pipe

To mark a class as a pipe and supply configuration metadata, apply the `@Pipe` decorator to the class. Use UpperCamelCase (the general convention for class names) for the pipe class name, and camelCase for the corresponding `name` string. Do not use hyphens in the `name`. For details and more examples, see Pipe names.

Use `name` in template expressions as you would for a built-in pipe.

- Include your pipe in the `declarations` field of the `NgModule` metadata in order for it to be available to a template. See the `app.module.ts` file in the example application (). For details, see NgModules.
- Register your custom pipes. The Angular CLI `ng generate pipe` command registers the pipe automatically.

Using the PipeTransform interface

Implement the `PipeTransform` interface in your custom pipe class to perform the transformation.

Angular invokes the `transform` method with the value of a binding as the first argument, and any parameters as the second argument in list form, and returns the transformed value.

Example: Transforming a value exponentially

In a game, you might want to implement a transformation that raises a value exponentially to increase a hero's power. For example, if the hero's score is 2, boosting the hero's power exponentially by 10 produces a score of 1024. Use a custom pipe for this transformation.

The following code example shows two component definitions:

- The `exponential-strength.pipe.ts` component defines a custom pipe named `exponentialStrength` with the `transform` method that performs

the transformation. It defines an argument to the `transform` method (`exponent`) for a parameter passed to the pipe.
- The `power-booster.component.ts` component demonstrates how to use the pipe, specifying a value (`2`) and the exponent parameter (`10`).

The browser displays the following:

Power Booster

Superpower boost: 1024

To examine the behavior the `exponentialStrength` pipe in the , change the value and optional exponent in the template.

{@a change-detection}

Detecting changes with data binding in pipes

You use data binding with a pipe to display values and respond to user actions. If the data is a primitive input value, such as `String` or `Number`, or an object reference as input, such as `Date` or `Array`, Angular executes the pipe whenever it detects a change for the input value or reference.

For example, you could change the previous custom pipe example to use two-way data binding with `ngModel` to input the amount and boost factor, as shown in the following code example.

The `exponentialStrength` pipe executes every time the user changes the "normal power" value or the "boost factor".

Angular detects each change and immediately runs the pipe. This is fine for primitive input values. However, if you change something *inside* a composite object (such as the month of a date, an element of an array, or an object property), you need to understand how change detection works, and how to use an `impure` pipe.

How change detection works

Angular looks for changes to data-bound values in a change detection process that runs after every DOM event: every keystroke, mouse move, timer tick, and server response. The following example, which doesn't use a pipe, demonstrates how Angular uses its default change detection strategy to monitor and update its display of every hero in the `heroes` array. The example tabs show the following:

- In the `flying-heroes.component.html (v1)` template, the `*ngFor` repeater displays the hero names.
- Its companion component class `flying-heroes.component.ts (v1)` provides heroes, adds heroes into the array, and resets the array.

Angular updates the display every time the user adds a hero. If the user clicks the **Reset** button, Angular replaces `heroes` with a new array of the original

heroes and updates the display. If you add the ability to remove or change a hero, Angular would detect those changes and update the display as well.

However, executing a pipe to update the display with every change would slow down your application's performance. So Angular uses a faster change-detection algorithm for executing a pipe, as described in the next section.

{@a pure-and-impure-pipes}

Detecting pure changes to primitives and object references

By default, pipes are defined as *pure* so that Angular executes the pipe only when it detects a *pure change* to the input value. A pure change is either a change to a primitive input value (such as `String`, `Number`, `Boolean`, or `Symbol`), or a changed object reference (such as `Date`, `Array`, `Function`, or `Object`).

{@a pure-pipe-pure-fn}

A pure pipe must use a pure function, which is one that processes inputs and returns values without side effects. In other words, given the same input, a pure function should always return the same output.

With a pure pipe, Angular ignores changes within composite objects, such as a newly added element of an existing array, because checking a primitive value or object reference is much faster than performing a deep check for differences within objects. Angular can quickly determine if it can skip executing the pipe and updating the view.

However, a pure pipe with an array as input might not work the way you want. To demonstrate this issue, change the previous example to filter the list of heroes to just those heroes who can fly. Use the `FlyingHeroesPipe` in the `*ngFor` repeater as shown in the following code. The tabs for the example show the following:

- The template (`flying-heroes.component.html (flyers)`) with the new pipe.
- The `FlyingHeroesPipe` custom pipe implementation (`flying-heroes.pipe.ts`).

The application now shows unexpected behavior: When the user adds flying heroes, none of them appear under "Heroes who fly." This happens because the code that adds a hero does so by pushing it onto the `heroes` array:

The change detector ignores changes to elements of an array, so the pipe doesn't run.

The reason Angular ignores the changed array element is that the *reference* to the array hasn't changed. Because the array is the same, Angular does not update the display.

One way to get the behavior you want is to change the object reference itself. Replace the array with a new array containing the newly changed elements, and

then input the new array to the pipe. In the preceding example, create an array with the new hero appended, and assign that to `heroes`. Angular detects the change in the array reference and executes the pipe.

To summarize, if you mutate the input array, the pure pipe doesn't execute. If you *replace* the input array, the pipe executes and the display is updated.

The preceding example demonstrates changing a component's code to accommodate a pipe.

To keep your component independent of HTML templates that use pipes, you can, as an alternative, use an *impure* pipe to detect changes within composite objects such as arrays, as described in the next section.

{@a impure-flying-heroes}

Detecting impure changes within composite objects

To execute a custom pipe after a change *within* a composite object, such as a change to an element of an array, you need to define your pipe as `impure` to detect impure changes. Angular executes an impure pipe every time it detects a change with every keystroke or mouse movement.

While an impure pipe can be useful, be careful using one. A long-running impure pipe could dramatically slow down your application.

Make a pipe impure by setting its `pure` flag to `false`:

The following code shows the complete implementation of `FlyingHeroesImpurePipe`, which extends `FlyingHeroesPipe` to inherit its characteristics. The example shows that you don't have to change anything else—the only difference is setting the `pure` flag as `false` in the pipe metadata.

`FlyingHeroesImpurePipe` is a good candidate for an impure pipe because the `transform` function is trivial and fast:

You can derive a `FlyingHeroesImpureComponent` from `FlyingHeroesComponent`. As shown in the following code, only the pipe in the template changes.

To confirm that the display updates as the user adds heroes, see the .

{@a async-pipe}

Unwrapping data from an observable

Observables let you pass messages between parts of your application. Observables are recommended for event handling, asynchronous programming, and handling multiple values. Observables can deliver single or multiple values of any type, either synchronously (as a function delivers a value to its caller) or asynchronously on a schedule.

For details and examples of observables, see the [Observables Overview](/guide/observables#using-observables-to-pass-values "Using observables to pass values"“).

Use the built-in `AsyncPipe` to accept an observable as input and subscribe to the input automatically. Without this pipe, your component code would have to subscribe to the observable to consume its values, extract the resolved values, expose them for binding, and unsubscribe when the observable is destroyed in order to prevent memory leaks. `AsyncPipe` is an impure pipe that saves boilerplate code in your component to maintain the subscription and keep delivering values from that observable as they arrive.

The following code example binds an observable of message strings (`message$`) to a view with the `async` pipe.

{@a no-filter-pipe}

Caching HTTP requests

To communicate with backend services using HTTP, the `HttpClient` service uses observables and offers the `HttpClient.get()` method to fetch data from a server. The asynchronous method sends an HTTP request, and returns an observable that emits the requested data for the response.

As shown in the previous section, use the impure `AsyncPipe` to accept an observable as input and subscribe to the input automatically. You can also create an impure pipe to make and cache an HTTP request.

Impure pipes are called whenever change detection runs for a component, which could be as often as every few milliseconds. To avoid performance problems, call the server only when the requested URL changes, as shown in the following example, and use the pipe to cache the server response. The tabs show the following:

- The `fetch` pipe (`fetch-json.pipe.ts`).
- A harness component (`hero-list.component.ts`) for demonstrating the request, using a template that defines two bindings to the pipe requesting the heroes from the `heroes.json` file. The second binding chains the `fetch` pipe with the built-in `JsonPipe` to display the same hero data in JSON format.

In the preceding example, a breakpoint on the pipe's request for data shows the following:

- Each binding gets its own pipe instance.
- Each pipe instance caches its own URL and data and calls the server only once.

The `fetch` and `fetch-json` pipes display the heroes in the browser as follows:

Heroes from JSON File

Windstorm Bombasto Magneto Tornado

Heroes as JSON: [{ “name”: “Windstorm”, “canFly”: true }, { “name”: “Bombasto”, “canFly”: false }, { “name”: “Magneto”, “canFly”: false }, { “name”: “Tornado”, “canFly”: true }]

The built-in JsonPipe provides a way to diagnose a mysteriously failing data binding or to inspect an object for future binding.

Pipes and precedence

The pipe operator has a higher precedence than the ternary operator (`?:`), which means `a ? b : c | x` is parsed as `a ? b : (c | x)`. The pipe operator cannot be used without parentheses in the first and second operands of `?:`.

Due to precedence, if you want a pipe to apply to the result of a ternary, wrap the entire expression in parentheses; for example, `(a ? b : c) | x`.

@reviewed 2021-09-15