In Angular, there are several types of pipes that you can use to transform data in templates. These pipes help you format, filter, and manipulate data before displaying it to users. Here are some common types of pipes in Angular:

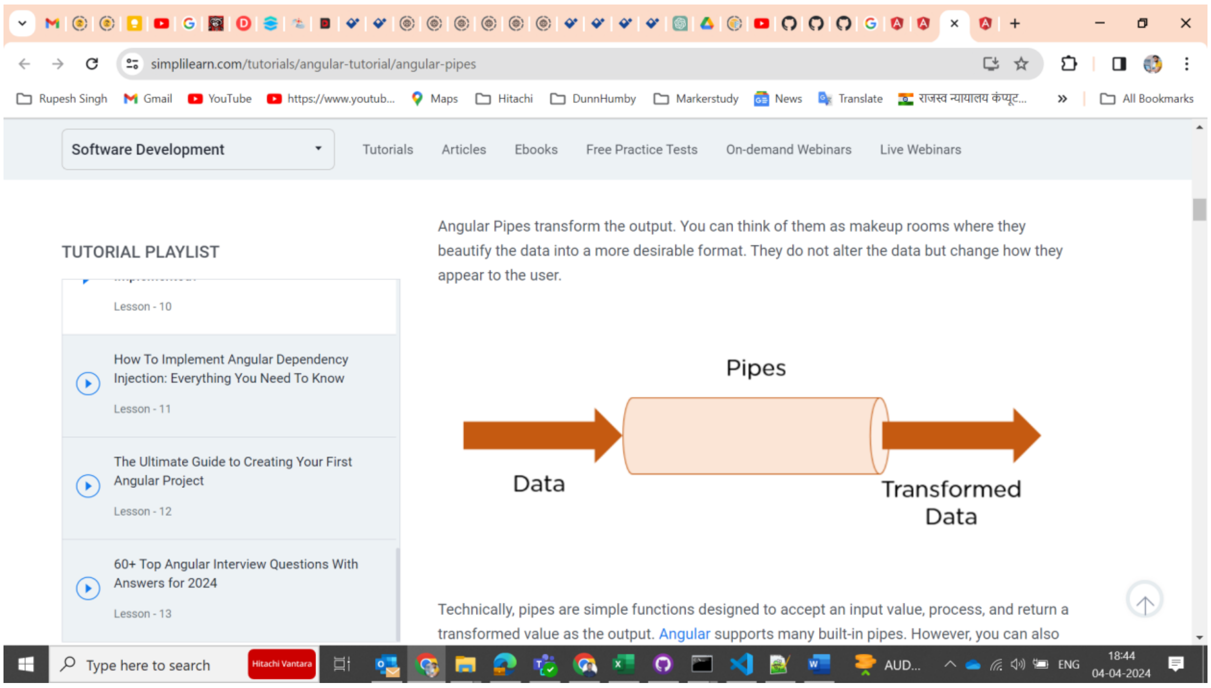
1. **Built-in Pipes:** Angular provides several built-in pipes for common data transformations, such as formatting dates, numbers, and currency, as well as transforming strings and arrays. Some examples include `DatePipe`, `DecimalPipe`, `CurrencyPipe`, `UpperCasePipe`, `LowerCasePipe`, `SlicePipe`, and `JsonPipe`.
2. **Custom Pipes:** You can create custom pipes to implement custom data transformations specific to your application's needs. Custom pipes are reusable and can be applied to any data in your application. To create a custom pipe, you need to implement the `PipeTransform` interface and define the `transform` method.
3. **Pure and Impure Pipes:** Pipes in Angular can be categorized as either pure or impure based on their behavior. Pure pipes are stateless and do not depend on external factors. They only recalculate when their input values change. Impure pipes, on the other hand, can have side effects and may recalculate on every change detection cycle. By default, Angular pipes are pure, but you can explicitly mark them as impure by setting the `pure` property to `false` in the `@Pipe` decorator.
4. **Async Pipes:** Async pipes are a special type of pipe in Angular used to handle asynchronous data streams, such as observables and promises. Async pipes automatically subscribe to an observable or promise, retrieve the emitted values, and update the view with the latest value. They manage the subscription lifecycle and automatically unsubscribe when the component is destroyed, preventing memory leaks.
5. **Chaining Pipes:** You can chain multiple pipes together in Angular to perform multiple transformations sequentially. Chaining pipes allows you to apply multiple data transformations to the same value in a template. For example, you can chain the `UpperCasePipe` and `SlicePipe` together to convert a string to uppercase and then extract a substring from it.

These are some of the common types of pipes in Angular. Using pipes, you can efficiently transform and format data in templates to meet your application's requirements.

Build in pipe examples

```
{{ today | date }}
```

{{ amount | currency : 'en-US' }}



Angular Pipes transform the output. You can think of them as makeup rooms where they beautify the data into a more desirable format. They do not alter the data but change how they appear to the user.

Technically, pipes are simple functions designed to accept an input value, process, and return a transformed value as the output. Angular supports many built-in pipes. However, you can also

Create the Custom Pipe:

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

@Pipe({
  name: 'capitalize'
})
export class CapitalizePipe implements PipeTransform {

  transform(value: string): string {
    if (!value) return ''; // Handle null, undefined, and empty strings
    return value.replace(/\b\w/g, firstChar => firstChar.toUpperCase());
  }
}
```

In this custom pipe:

- We import `Pipe` and `PipeTransform` from `@angular/core`.
- We decorate the class with `@Pipe` and provide a `name` property to define the pipe's name (`capitalize`).
- We implement the `PipeTransform` interface and provide the `transform` method, which takes the input value (a string) and transforms it by capitalizing the first letter of each word using a regular expression.

2. **Use the Custom Pipe in a Component Template:**

Assuming you have a component template (`app.component.html`), you can use the `capitalize` pipe as follows:

`app.component.html`

```
<p>{{ 'hello world' | capitalize }}</p>
```

This will output:

```
Hello World
```

Register the Custom Pipe:

```
@NgModule({
  declarations: [
    AppComponent,
    CapitalizePipe // Declare the custom pipe
  ],
```

Using the `async` Pipe with Observables:

Defi:

The `async` pipe subscribes to an `Observable` or `Promise` and returns the latest value it has emitted. When a new value is emitted, the `async` pipe marks the component to be checked for changes. When the component gets destroyed, the `async` pipe unsubscribes automatically to avoid potential memory leaks. When the reference of the expression changes, the `async` pipe automatically unsubscribes from the old `Observable` or `Promise` and subscribes to the new one.

It's also possible to use `async` with Observables. The example below binds the `time` Observable to the view. The Observable continuously updates the view with the current time.

```
export class AppComponent {
  title = 'callbackfunction';
  time = new Observable<string>((observer: Observer<string>) => {
    setInterval(() => observer.next(new Date().toString()), 1000);
  });
```

```
greeting: Promise<string> | null = null;
  arrived: boolean = false;

  private resolve: Function | null = null;


  fetchedData: ProductItem[] = [];
  error: any;
  newData: any;
  constructor(private dataService: DataService) {
    this.reset();
   }
   reset() {
    this.arrived = false;
    this.greeting = new Promise<string>((resolve, reject) => {
      this.resolve = resolve;
    });
  }
  clicked() {
    if (this.arrived) {
      this.reset();
    } else {
      this.resolve!('hi there!');
      this.arrived = true;
    }
  }
```

Chaining pipes in Angular allows you to apply multiple transformations to data sequentially in the template. This can be useful for formatting or manipulating data in various ways before displaying it to the user. Here are some examples of chaining pipes in Angular:

```
<!-- Example: Chaining date and uppercase pipes -->
```

```
<!-- Example: Chaining date and uppercase pipes -->
```

```
<p>{{ today | date:'shortDate' | uppercase }}</p>

<!-- Output: e.g., 10/13/21 (assuming today is October 13, 2021) -->
```

```
<!-- Output: e.g., 10/13/21 (assuming today is October 13, 2021) -->
```

Chaining Custom Pipes:

Suppose we have a custom pipe called `truncate` that truncates a string to a specified length:

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


@Pipe({
  name: 'truncate'
})
export class TruncatePipe implements PipeTransform {
  transform(value: string, length: number): string {
    if (!value) return '';
    return value.length > length ? value.substring(0, length) + '...' : value;
  }
}
```

Output:

```
<!-- Example: Chaining custom truncate and uppercase pipes -->
<p>{{ longText | truncate:20 | uppercase }}</p>

<!-- Output: e.g., "THIS IS A LONG TEXT..." -->
```