Angular JS

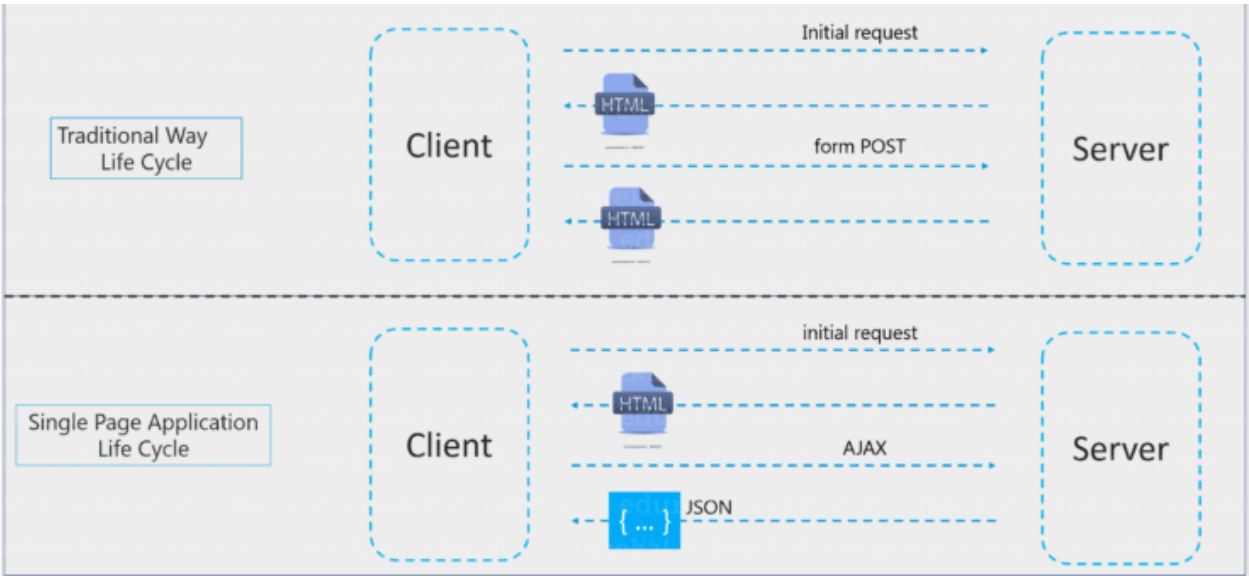
Angular is basically an open-source, JavaScript-based client-side framework that helps us to develop a web-based application. Angular is known as one of the best frameworks for developing any Single Page Application or SPA application.

Single-Page Applications (or ***SPA's***) are applications that are accessed via a web browser like other websites but offer more dynamic interactions resembling native mobile and desktop apps.

The most notable difference between a regular website and SPA is the reduced amount of page refreshes.

SPAs have a heavier usage of AJAX- a way to communicate with back-end servers without doing a full page refresh to get data loaded into our application.

As a result, the process of rendering pages happens mostly on the client-side.



To design any Angular app we are going to use modules, components, templates, metadata, data binding, directives, services, and dependency injection.

Angular libraries

Angular gives us a collection of JavaScript modules (library modules) that provide various functionalities.

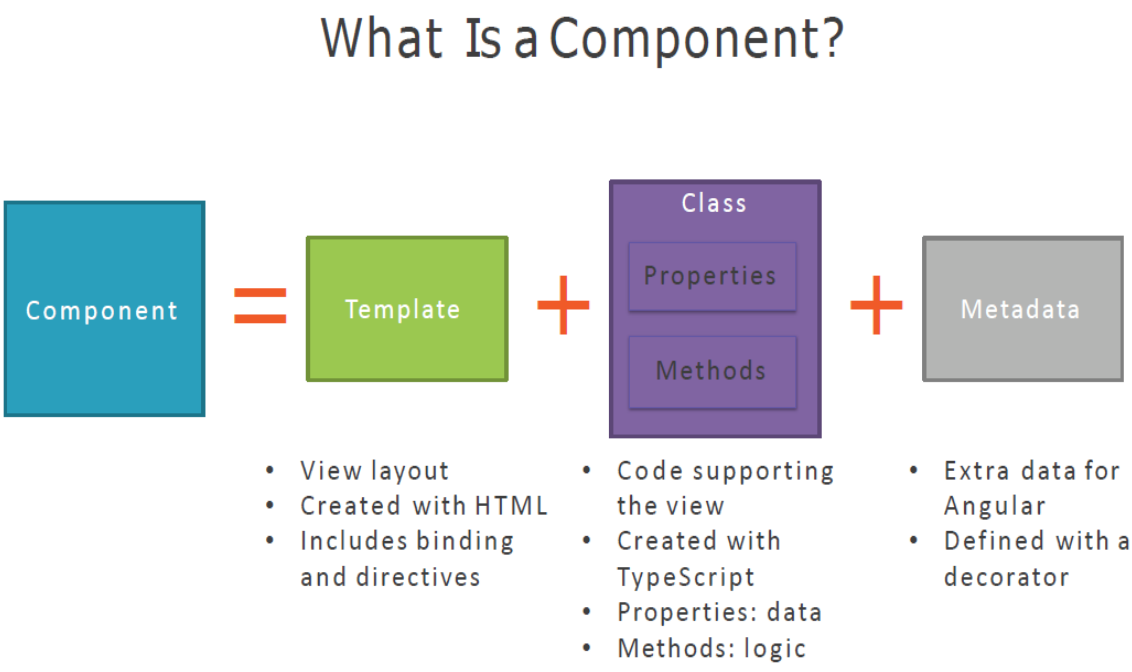
Each Angular library has @angular prefix, like *@angular/core*, *@angular/compiler*, *@angular/compiler-cli*, *@angular/http*, *@angular/router*. You can install them using the **npm** package manager and import parts of them with JavaScript import statements.

Example : import { Component } from '@angular/core';

Components:

Angular Components (also known as Angular Web Components or Angular UI Components), are reusable pieces of code that define views. The view is a set of elements comprising distinct portions of the screen.

In Angular applications, we use individual components to define distinct sections and control various aspects of the application.



Decorator :

Decorators are mainly used to add meta data to class.Decorators are represented with @ symbol.

For **components**, we are going to use @Component()

Example:

```
@Component({
selector: 'pm-app', template: `
<div><h1>{{pageTitle}}</h1>
<div>My First Component</div>
</div>
})
export class AppComponent {
pageTitle: string = 'Acme Product Management';
}
```

Life Cycle of Angular Components :

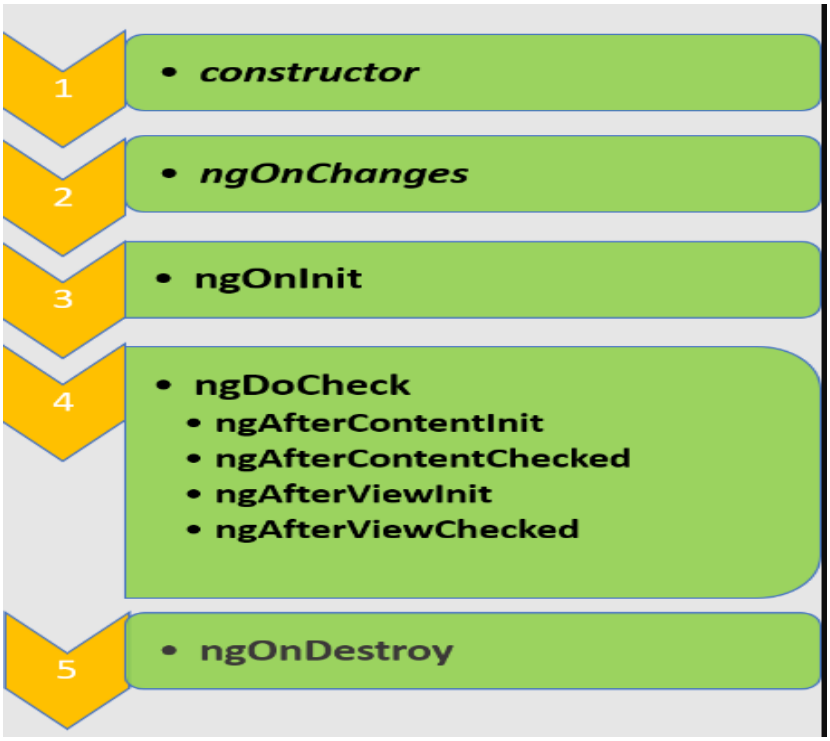
Every Angular component instance has a lifecycle associated to it. The component lifecycle begins when the component is initialized and ends at the destruction of the component.

Every component goes through 8 different phases (or stage) in its lifecycle.

The first stage of the component's lifecycle starts when Angular instantiates the component class and renders the component view along with its child views.

Followed by change detection algorithm, as Angular checks to see when data-bound properties change, and updates both the view and the component instance as needed.

The lifecycle ends when Angular destroys the component instance and removes its rendered template from the DOM.



HOOK METHOD	Use and Timing
ngOnChanges()	This hook executes every time one or more data-bound input properties within the component change. The hook receives a SimpleChange object; the SimpleChange object contains the current and previous values of the bound property.
	If the component has bound inputs, ngOnChanges() gets called before ngOnInit()
	This event is mostly used to initialize data in a component.
ngOnInit()	This lifecycle hook is triggered when Angular first displays the component's data-bound properties or when the component has been initialized. In other words, this event is called only once just after the first ngOnChanges() events.

	In case there are no template-bound inputs, ngOnChanges() will not execute but ngOnInit() will execute when component is initialized.
ngDoCheck()	This hook is called when the change detection mechanism detects some change in the input properties of a component. Moreover, this lifecycle hook is executed immediately after ngOnInit() on the first run, and after ngOnChanges() on every change detection run
	This hook is useful for implementing your own custom change detection logic for the target component.
ngAfterContentInit()	This hook is called only for the first time just after the ngDoCheck() hook when all the bindings of the component need to be checked.
	This lifecycle hook method executes when Angular performs any content projection within the component view.
	This method is fundamentally linked to the child components initialization.
ngAfterContentChecked()	This lifecycle hook is called whenever the component is checked by the Angulars in built change detection mechanism.
	The ngAfterContentChecked hook gets called after ngAfterContentInit() method and after every subsequent execution of ngDoCheck().
	This method is fundamentally linked to the child components' initialization.
ngAfterViewInit()	This hook executes when the component’s view and child views are fully initialized.
	This hook gets called only once after the first ngAfterContentChecked().
ngAfterViewChecked()	This method is just called after the ngAterViewInit() and every subsequent call of ngAfterContentChecked() hook.
	It is executed every time Angular checks the view of the host component’s view and child view, or the view that contains the directive.
	This lifecycle hook is extremely useful when the component is waiting for some value from its child components.
ngOnDestroy()	At last, the ngOnDestroy() lifecycle hook gets triggered

	before the target component is finally destroyed and removed from the DOM.
	It is used to unsubscribe the observables and detach event handlers and prevent memory leaks.

Component Interaction :

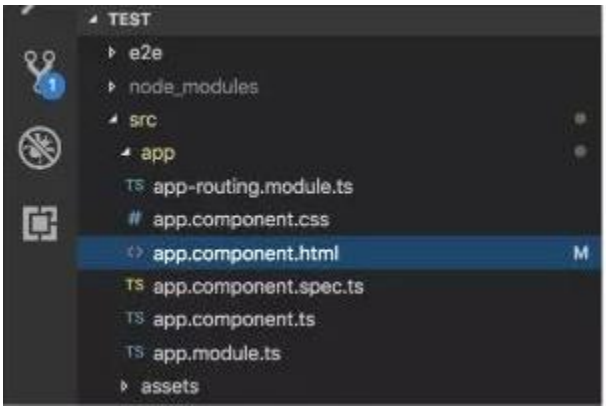
In an Angular application, components interact with each other to share data to and from different components. There are four different ways of sharing/ sending/ interacting data among different components.

- Parent to Child: Sharing Data via Angular Components Input
- Child to Parent: Sharing Data via ViewChild
- Child to Parent: Sharing Data via Output and EventEmitter
- Child to Parent: Sharing Data using local variable
- Unrelated Components: Sharing Data with a Service

Templates:

You associate the component's view with its companion **template**. A template is nothing but a form of HTML tags that tells Angular about how to render the component.

A template looks like regular HTML, except for a few differences.



In the above screenshot, you can see that there are a total of 6 files as of now inside the src/app folder.

- **app.module.ts:** This is the main module file for the project. Each Angular project is divided into modules to make the project management easy. By default, the whole project is a part of a single module but you can always create more modules. A module is a logical chunk of project resources that works independently or in combination with other modules. This file contains a module called AppModule.
- **app.component.html:** This file is the template file for the one and only component in our project as of now. This component is called the AppComponent. The extension html indicated that this file is an HTML document and will define the view of the component.
- **app.component.css:** This file work in combination with the HTML template and adds CSS styles to the HTML elements defined in that template. This only only contains CSS classes and styles.
- **app.component.ts:** This is the main class file for the AppComponent component. This file contains the TypeScript class for the component and all the business logic for this component goes inside this file.
- **app.component.spec.ts:** This file is for unit testing the AppComponent component. Let's not worry about this for now. Feel free to delete the file if it bothers you too much.
- **app-routing.module.ts:** This is another module file for the project (just like the app.module.ts that contains the module AppModule) that handles routing or navigation for the whole project. It defines how different components will be presented to the user as the user navigates within the application.

So, it is fairly evident from the above description of the files that a component is composed of 3 files -

1. The HTML Template

2. The TS Class
3. The CSS Styles

Metadata:

Metadata tells Angular how to process a class. To tell Angular that Course Component is a component, **metadata** is attached to the class. In TypeScript, you attach metadata by using a **decorator**.

Data Binding is a way to synchronize the data between the model and view components automatically. AngularJS implements data-binding that treats the model as the single-source-of-truth in your application & for all the time, the view is a projection of the model. Unlike React, angular supports two-way binding.

In this way, we can make the code more loosely coupled. Data binding can be categorized into 2 types, ie., One-way Binding & Two-way Binding.

One-way Binding: This type of binding is unidirectional, i.e. this binds the data flow from either component to view(DOM) or from the view(DOM) to the component.

There are various techniques through which the data flow can be bind from component to view or vice-versa.

If the *data flow from component to view(DOM)*, then this task can be accomplished with the help of **String Interpolation** & **Property Binding.**

Directives:

HTML doesn't support conditional attributes (if, if-else, switch, etc.), loop statements (loops), and other attributes. Angular Directives are used to extend the power of existing Html markup by adding new attributes or tags. They change the appearance or behavior of a DOM element.

There are three types of directives in Angular:

Component Directives: These are the most common directives in Angular, which are responsible for creating reusable UI components. Components are directives with a template. They have a view associated with them and can contain logic and data-binding.

Attribute Directives: These directives are used to modify the appearance or behavior of an element.

They are applied as attributes in HTML tags and can change the element's properties, apply styles, or add/remove classes. Angular provides some built-in attribute directives like ngClass, ngStyle, and ngModel.

Structural Directives: These directives change the structure of the DOM by adding or removing elements. They are used to conditionally render elements based on certain conditions. Examples of structural directives in Angular 8 include ngIf, ngFor, and ngSwitch.

Pipes:

Pipes are referred as filters. It helps to transform data and manage data within interpolation, denoted by {{ | }}. It accepts data, arrays, integers and strings as inputs which are separated by '|' symbol.

In Angular, pipes are a feature that allows you to transform and format data in templates. Pipes are used to modify the appearance or behavior of values before they are displayed to the user. They can be applied to data binding expressions within double curly braces (`{{}}`) or in the `ngModel` directive.

Angular provides several built-in pipes, and you can also create custom pipes to meet your specific requirements. Here are some commonly used built-in pipes in Angular:

1. DatePipe: Used to format dates. It provides various options for displaying dates in different formats.

Example: `{{ currentDate | date:'short' }}`

2. UpperCasePipe: Converts a string to uppercase.

Example: `{{ 'hello' | uppercase }}`

3. LowerCasePipe: Converts a string to lowercase.

Example: `{{ 'WORLD' | lowercase }}`

4. DecimalPipe: Formats a number as a decimal according to locale rules.

Example: `{{ price | number:'1.2-2' }}`

5. CurrencyPipe: Formats a number as currency according to locale rules.

Example: `{{ price | currency:'USD':true }}`

6. PercentPipe: Formats a number as a percentage.

Example: `{{ discount | percent }}`

7. SlicePipe: Extracts a portion of a string or an array.

Example: `{{ text | slice:0:10 }}`

To create a custom pipe, you can use the `@Pipe` decorator .

Forms:

Forms are used to handle user input data. Angular 8 supports two types of forms. They are **Template driven forms** and **Reactive forms**.

Both approaches are used to collect user input events from the view, validate the user input, create a form model and data model to update, and provide a way to track changes.

Template-driven Forms

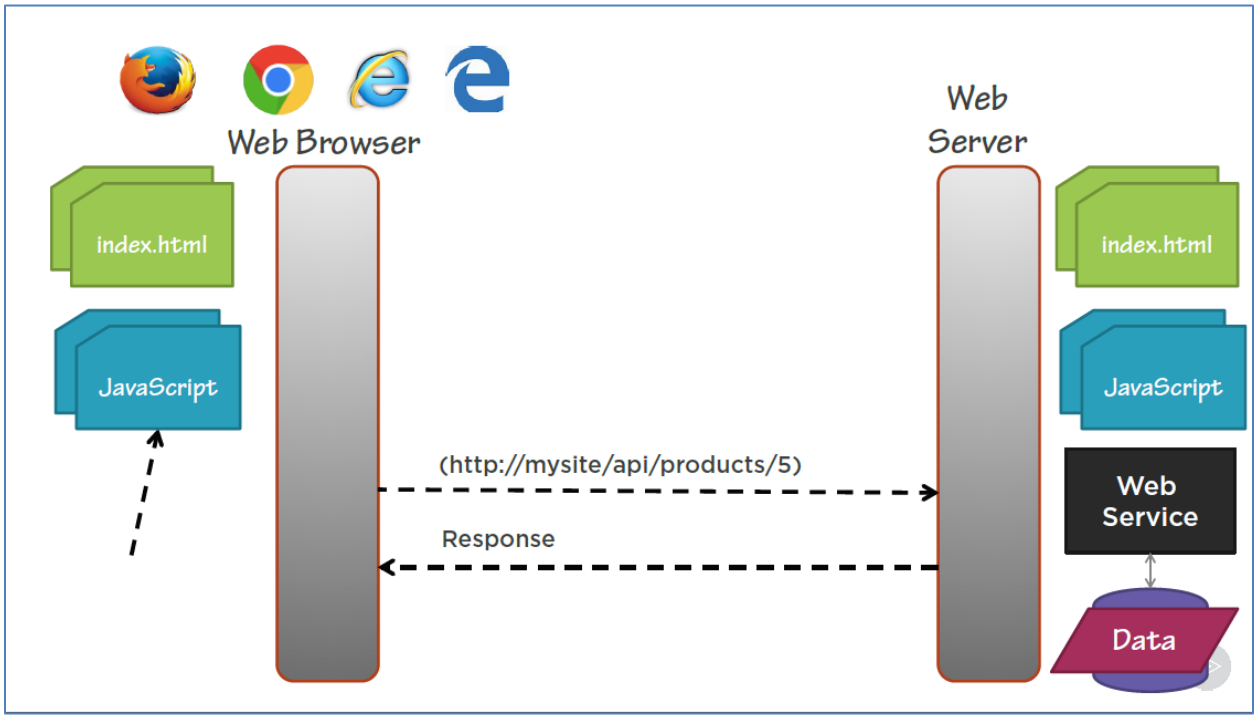
- Template-driven forms are best if you want to add a simple form to your application. **For example:** email list signup form.

- Template-driven forms are easy to use in the application but they are not as scalable as Reactive forms.
- Template-driven forms are mainly used if your application's requires a very basic form and logic. It can easily be managed in a template.

Reactive Forms

- Reactive forms are more robust.
- Reactive forms are more scalable, reusable, and testable.
- They are most preferred to use if forms are a key part of your application, or your application is already built using reactive patterns. In both cases, reactive forms are best to use.

HttpClient:



Angular provides a powerful module called HttpClient for making HTTP requests.

This module simplifies the process of sending requests to a server and handling the response. Here's an overview of how to work with HTTP requests in Angular:

1. Import the HttpClientModule: Before you can start making HTTP requests, you need to import the HttpClientModule in your Angular module. You can typically find this in the app.module.ts file.

```typescript
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [HttpClientModule],
  // ...
})
export class AppModule { }
```

2. Inject the HttpClient service: In your component or service, inject the HttpClient service so that you can use it to make HTTP requests.

```typescript
import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) { }
```

3. Make a GET request: To make a GET request, use the `get()` method of the HttpClient service. The `get()` method returns an Observable that you can subscribe to in order to receive the response.

```typescript
```

```
this.http.get('https://api.example.com/data').subscribe(
  (response) => {
    // Handle the response
    console.log(response);
  },
  (error) => {
    // Handle errors
    console.error(error);
  }
);
```

4. Make a POST request: To make a POST request, use the `post()` method of the HttpClient service. You can also pass data along with the request.

```typescript
const data = { name: 'John', age: 25 };
this.http.post('https://api.example.com/data', data).subscribe(
  (response) => {
    // Handle the response
    console.log(response);
  },
  (error) => {
```

```
  // Handle errors
  console.error(error);
 }
);
```

5. Handling the response: The response received from the server can be in various formats such as JSON, text, or blobs. By default, HttpClient automatically parses JSON responses. If the response is not in JSON format, you can specify the response type using the `responseType` option.

```typescript
this.http.get('https://api.example.com/data', { responseType: 'text' }).subscribe(
 (response) => {
  // Handle the response
  console.log(response);
 },
 (error) => {
  // Handle errors
  console.error(error);
 }
);
```

Routing:

In Angular 8, you can use the Angular Router module to handle routing and navigation within your application. The Angular Router provides a way to map URLs to different components, allowing you to navigate between different views.

Here are the basic steps to set up routing and navigation in Angular 8:

1. Install the Angular Router module:

```
npm install @angular/router
```

2. Set up your routes:

In your app module or a separate routing module, define your routes using the `RouterModule.forRoot()` method. Each route consists of a path and the component that should be displayed when the path is accessed. For example:

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home.component';
import { AboutComponent } from './about.component';

const routes: Routes = [
  { path: '', redirectTo: '/home', pathMatch: 'full' },
  { path: 'home', component: HomeComponent },
```

```
  { path: 'about', component: AboutComponent },
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

3. Add the `<router-outlet></router-outlet>` directive to your app component template:

The `<router-outlet></router-outlet>` directive is used to mark the location where the routed views will be displayed in your application's template.

4. Add navigation links:

In your application's templates, you can use the `routerLink` directive to create links that navigate to different routes. For example:

```html
<a routerLink="/home">Home</a>
<a routerLink="/about">About</a>
```

Alternatively, you can also use the `routerLink` directive with an array parameter to specify dynamic routes or provide route parameters:

```html
```

<a [routerLink]="['/product', productId]">Product Details

```

5. Implement navigation programmatically:

In your component classes, you can import the `Router` service from `@angular/router` and use its methods to navigate programmatically. For example:

```typescript
import { Router } from '@angular/router';

constructor(private router: Router) { }

goToAboutPage() {
  this.router.navigate(['/about']);
}
```
```