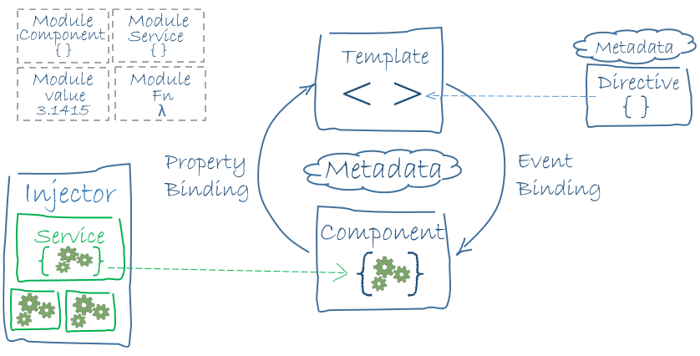
**Concept of Angular 6**



Angular framework use component to develop the application as in component we have two different parts one is html part attached with **templateUrl** and another with it’s corresponding typescript part that will take care the event ,generated from the UI part and how the two parts connected with each other ,as i told using **templateUrl .**

**Working flow of Angular 6 App:-**

1>index.html loads in browser

2>control goes to main.js file

3>there in main.js file we find component that will hold all your application’s components as a container in platformBrowserDynamic().bootstrapModule(AppModule);

here AppModule is the class that holds all your application’s components.

Different parts of An Angular 6 App:---

### Angular libraries



Angular ships as a collection of JavaScript modules. You can think of them as library modules.

Each Angular library name begins with the @angular prefix.

You install them with the npm package manager and import parts of them with JavaScript import statements.

## 1> Modules :--

Angular apps are modular and Angular has its own modularity system called NgModules.Every Angular app has at least one NgModule class, the root module, conventionally named AppModule.

While the root module may be the only module in a small application, most apps have many more feature modules, each a cohesive block of code dedicated to an application domain, a workflow, or a closely related set of capabilities.

An NgModule, whether a root or feature, is a class with an @NgModule decorator.

A Introduction:-------------

NgModule is a decorator function that takes a single metadata object whose properties describe the module. The most important properties are:

* declarations - the view classes that belong to this module. Angular has three kinds of view classes: components, directives, and pipes.

(In simple language the components that we as a developer creates not the routing module e.g in our projects we have –login,home,emp etc)

* exports - the subset of declarations that should be visible and usable in the component templates of other modules.

(In simple language if you need to use any class that means the typescript code of one class in another class ,you need to use export keyword at the time of the class declaration so that in destination class you can import the class as we did in project with service,loginMst class etc.)

* imports - other modules whose exported classes are needed by component templates declared in this module.

(As above but here in destination we use import to load the required class)

* providers - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.

(In simple language if we create a service class that with @injectable then to register the service with angular 6 app either place within provider decorater in app.module.ts file if you need to import it any component within the app else put a provider decorater in a component itself just below templateUrl and place the service class there then the service will only be available with in the component only not to other components )

@Component({

  selector: 'my-app',

  templateUrl: 'app/empList.html',

  providers:[EmpService]

})

export class empComponent implements OnInit {

}

* bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

Here we supply the first component class name to be loaded first when my application firstly loaded.

It shuld be placed in app.module.ts file.

bootstrap:    [ AppComponent ]

Here's a simple root module:

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

@NgModule({

imports: [ BrowserModule ],

providers: [ Logger ],

declarations: [ AppComponent ],

exports: [ AppComponent ],

bootstrap: [ AppComponent ]

})

export class AppModule { }

**Component:--**

A component controls a patch of screen called a view.

You define a component's application logic—what it does to support the view—inside a class. The class interacts with the view through an API of properties and methods.

Angular creates, updates, and destroys components as the user moves through the application. Your app can take action at each moment in this lifecycle through optional lifecycle hooks, like ngOnInit() etc.

## **Templates:--**

You define a component's view with its companion template. A template is a form of HTML that tells Angular how to render the component.

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

(In simple language we put the HTML code in template or templateUrl)

**Metadata:--**

## Metadata tells Angular how to process a class.

* 1. Here is the @Component decorator, which identifies the class immediately below it as a component class.

The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

Here are a few of the most useful @Component configuration options:

* **selector**: CSS selector that tells Angular to create and insert an instance of this component where it finds a <my-app> tag in parent HTML. For example, if an app's HTML contains <my-app></my-app>, then Angular inserts an instance of the view between those tags.
* **templateUrl**: module-relative address of this component's HTML template.
* **Template**: Direct coding of this component's HTML .
* The template is a multi-line string within ECMAScript 2015 backticks (`). The backtick (`)—which is not the same character as a single quote (')—allows you to compose a string over several lines, which makes the HTML more readable.
* providers: array of dependency injection providers for services that the component requires. This is one way to tell Angular that the component's constructor requires a HeroService so it can get the list of heroes to display.

The metadata in the @Component tells Angular where to get the major building blocks you specify for the component.

The template, metadata, and component together describe a view.

Apply other metadata decorators in a similar fashion to guide Angular behavior. @Injectable, @Input, and @Output are a few of the more popular decorators.

## Directives



Angular templates are dynamic. When Angular renders them, it transforms the DOM according to the instructions given by directives.

A directive is a class with a @Directive decorator. A component is a directive-with-a-template; a @Component decorator is actually a @Directive decorator extended with template-oriented features.

While a component is technically a directive, components are so distinctive and central to Angular applications that this architectural overview separates components from directives.

Two other kinds of directives exist: structural and attribute directives.

They tend to appear within an element tag as attributes do, sometimes by name but more often as the target of an assignment or a binding.

## Data binding

Without a framework, you would be responsible for pushing data values into the HTML controls and turning user responses into actions and value updates. Writing such push/pull logic by hand is tedious, error-prone, and a nightmare to read as any experienced jQuery programmer can attest.



Angular supports data binding, a mechanism for coordinating parts of a template with parts of a component. Add binding markup to the template HTML to tell Angular how to connect both sides.

As the diagram shows, there are four forms of data binding syntax. Each form has a direction — to the DOM, from the DOM, or in both directions.

## Services



Service is a broad category encompassing any value, function, or feature that your application needs.

Almost anything can be a service. A service is typically a class with a narrow, well-defined purpose. It should do something specific and do it well.

## Dependency injection



Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies it requires. Most dependencies are services. Angular uses dependency injection to provide new components with the services they need.

Angular can tell which services a component needs by looking at the types of its constructor parameters.

When Angular creates a component, it first asks an injector for the services that the component requires.

An injector maintains a container of service instances that it has previously created. If a requested service instance is not in the container, the injector makes one and adds it to the container before returning the service to Angular. When all requested services have been resolved and returned, Angular can call the component's constructor with those services as arguments. This is dependency injection.

* Dependency injection is wired into the Angular framework and used everywhere.
* The injector is the main mechanism.
  + An injector maintains a container of service instances that it created.
  + An injector can create a new service instance from a provider.
* A provider is a recipe for creating a service.
* Register providers with injectors.

**How the app starts**

The CSS selector in the @Component decorator specifies an element named <my-app>. That element is a placeholder in the body of your index.html file:

<body>

<my-app>loading...</my-app>

</body>

When you bootstrap with the AppComponent class (in main.ts), Angular looks for a <my-app> in the index.html, finds it, instantiates an instance of AppComponent, and renders it inside the <my-app> tag.

## Template inline or template file?

You can store your component's template(UI Code) in one of two places. You can define it inline using the template property, or you can define the template in a separate HTML file and link to it in the component metadata using the @Component decorator's templateUrl property.

The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy. Here the app uses inline HTML because the template is small and the demo is simpler without the additional HTML file.

In either style, the template data bindings have the same access to the component's properties.