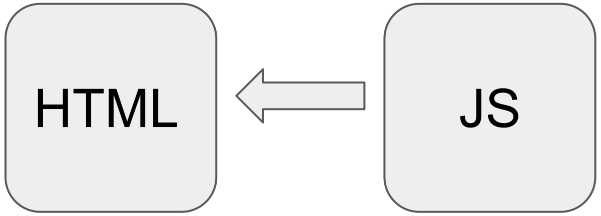
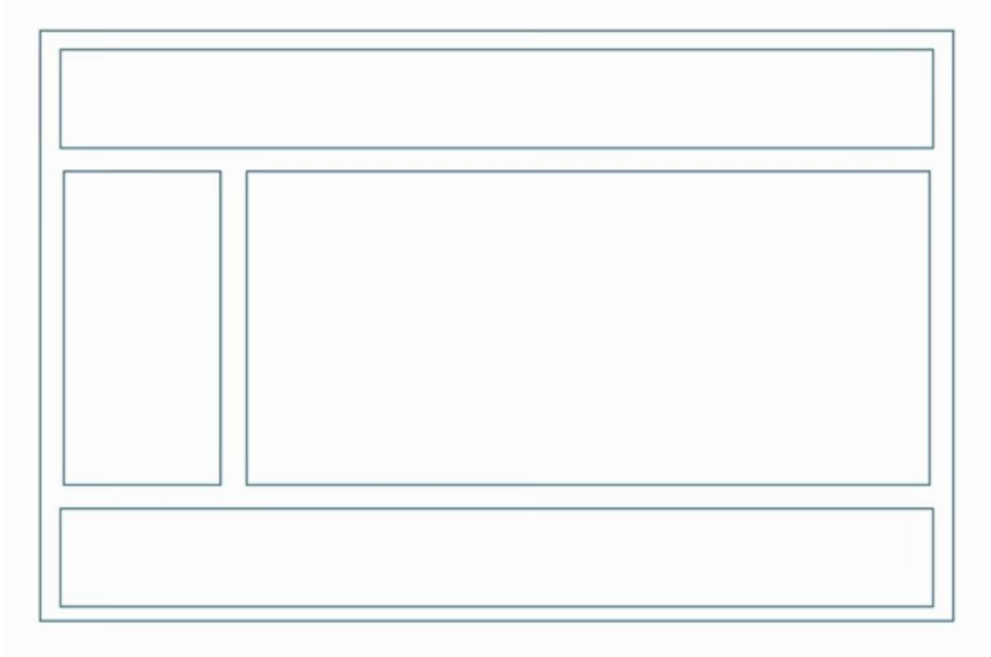
ANGULAR

Prerequisites

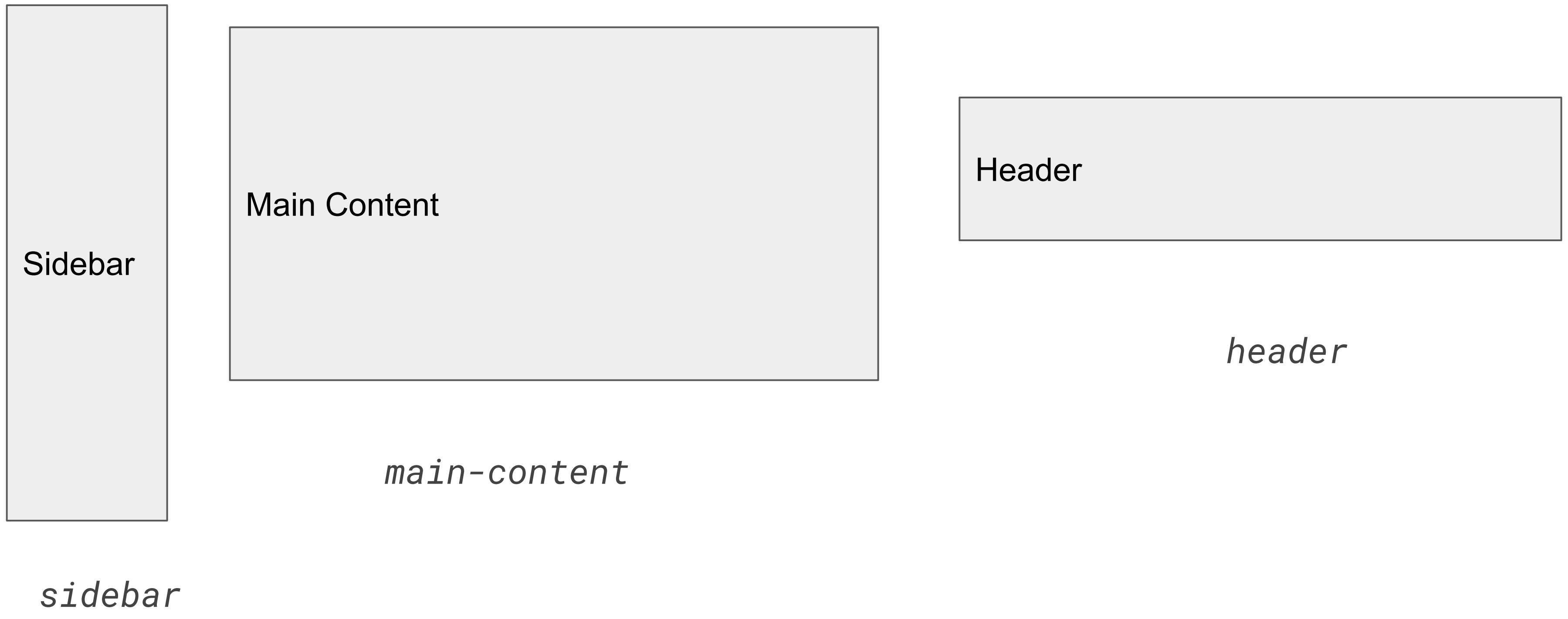
- HTML CSS
- Javascript / Typescript

Traditional vs The component based model





Traditional vs The component based model



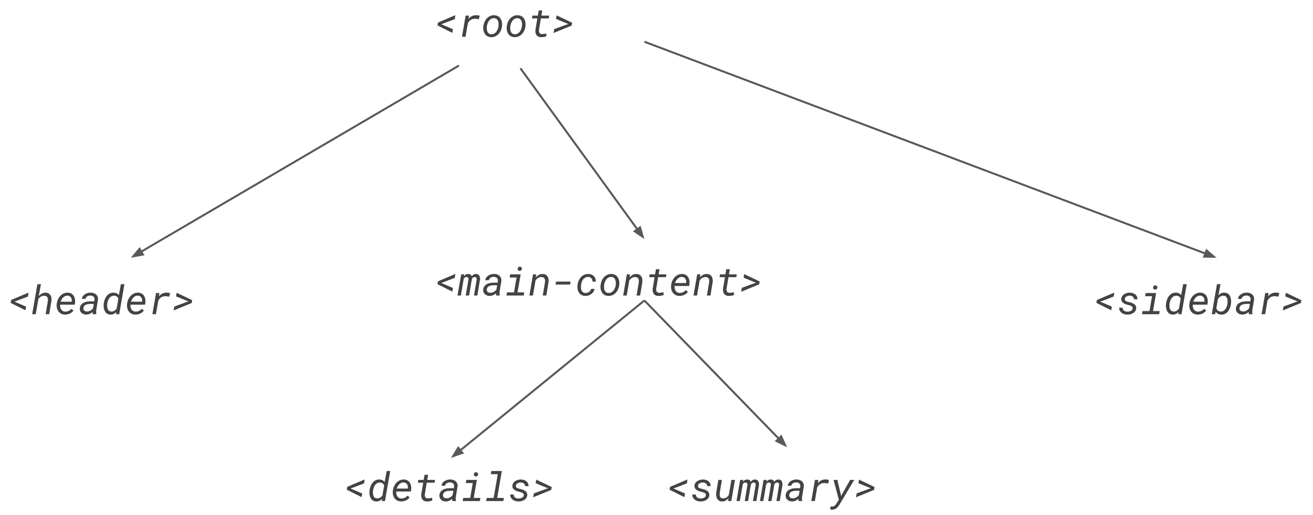
Traditional vs The component based model

```
<header>

<sidebar>

 <main-content>
```

Traditional vs The component based model



What is Angular

- Cross Platform Framework
    - Progressive Web Apps | Native Mobile | Mobile Web
- **TypeScript**-based
- Led by the Angular Team at Google

- Native vs Hybrid ?

Reference : https://angular.io/docs

Index

Difference : AngularJS & Angular

- Remember : Both are different
- 2010 : AngularJS
  - 2012–14 frameworks like ember.js and react.js (developed by Facebook) popped in with a better benchmark results and performance.
- 2016 September Angular2
  - Complete rewrite of AngularJS.
  - Good in terms of performance
  - Provides you an end-to-end solution, from testing to hybrid-mobile-apps to animations.
  - Entire angular app development process is suitable for large teams and large projects
- Typescript
  - Typescript is a typed superset of JavaScript that compiles to plain JavaScript
  - **Angular is written in TypeScript.**
  - **Angular implements core and optional functionality as a set of TypeScript libraries that you import into your apps**

Getting Started

Prerequisites

1. Node.js
2. npm package manager *(npm is installed with Node.js)*

Step 1: Install the Angular CLI

Step 2: Create a workspace and initial application

Step 3: Serve the application

Angular CLI A command line interface for Angular

- `npm install -g @angular/cli`

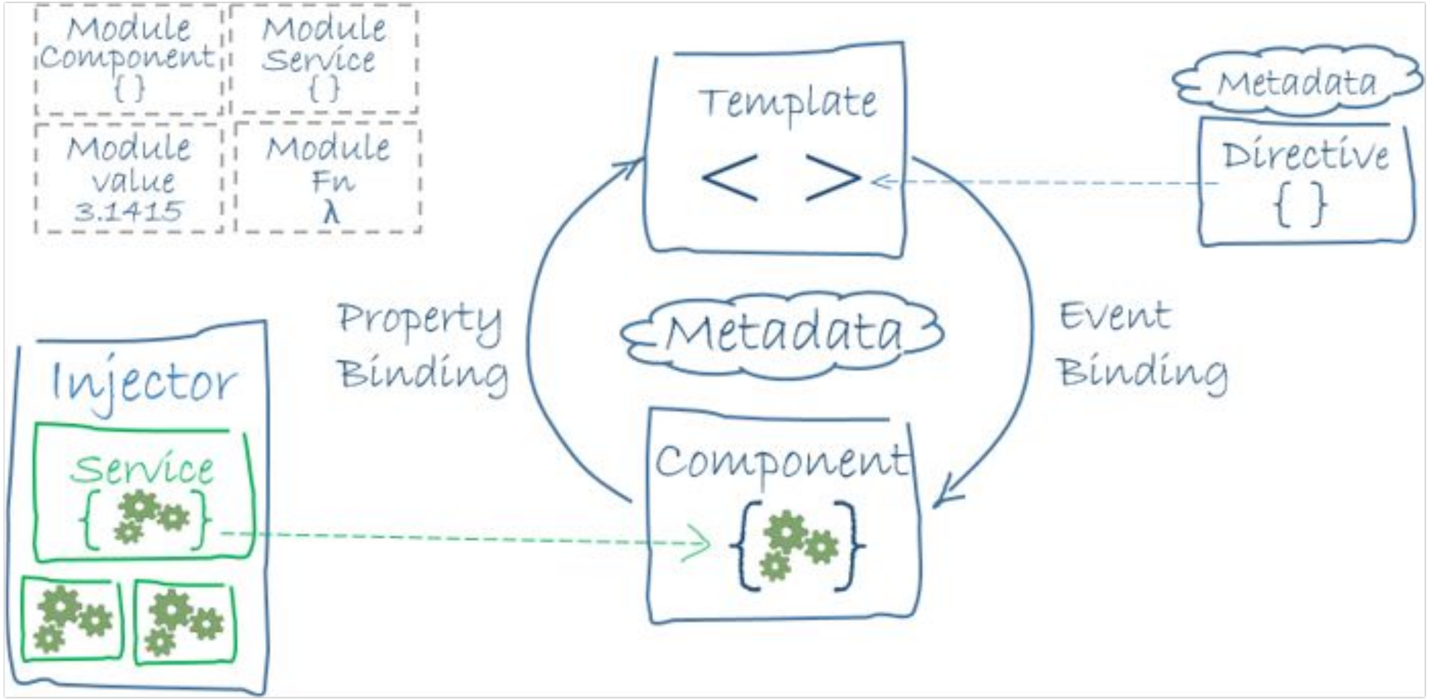
- `ng new my-dream-app`
- `cd my-dream-app`

- `ng serve --host <ip address> --port <port number>`

- `ng generate <service/component/….> <name>`

Index

Angular Architecture



building blocks of an Angular application:

1. Modules
2. Components
3. Templates
4. Metadata
5. Data binding
6. Directives
7. Services
8. Dependency injection

Components

- A component **controls a patch of screen** called a ***view***.
- We define 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**,

Templates

A **template** is a form of HTML that tells Angular how to render the component.

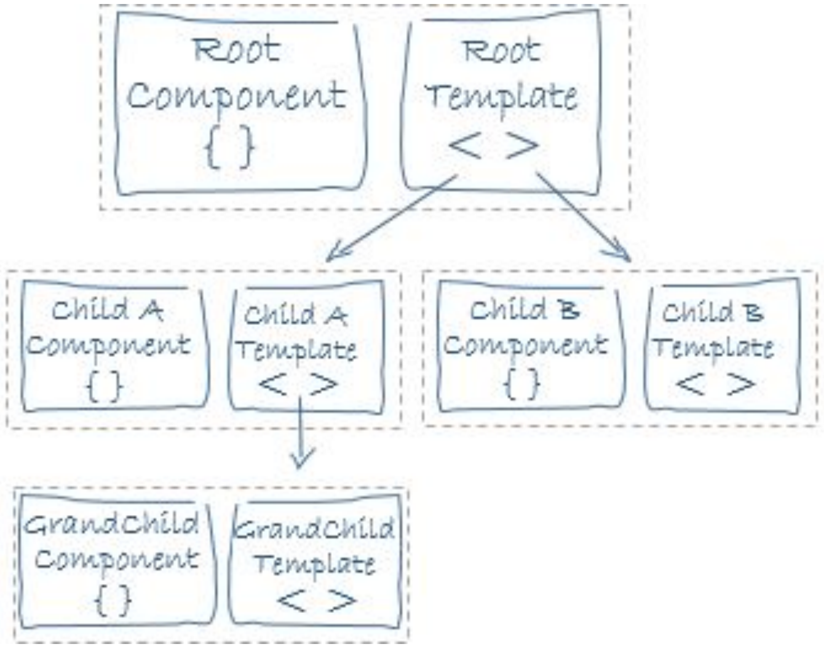
Uses **typical HTML** elements like <h2> and <p>

Angular's template syntax.
With code like *ngFor, {{hero.name}}, (click), [hero], and <hero-detail>

<hero-detail> is a **Custom component** tag

~ Custom components

Custom components mix seamlessly with native HTML in the same layouts.

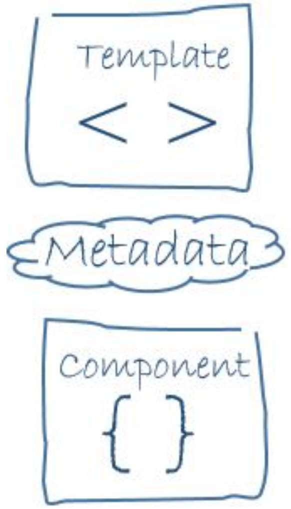


Metadata (With Decorators)

Decorators are functions that modify JavaScript classes.

Example

```
@Component({
        selector:    'hero-list',
        templateUrl: './hero-list.component.html',
        styleUrls: ['./app.component.css']
        …. })
```



@Component decorator will indicate

- the **selector** (the html tag name), → <hero-list>

  Tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML.
- the html **templateUrl** (as a string or as a path),
- the css **styleUrls** it encapsulates (as an array of css definitions or as a path), etc.

Metadata (With Decorators)

Angular has many **decorators that attach metadata to classes** so that it knows what those classes mean and how they should work.

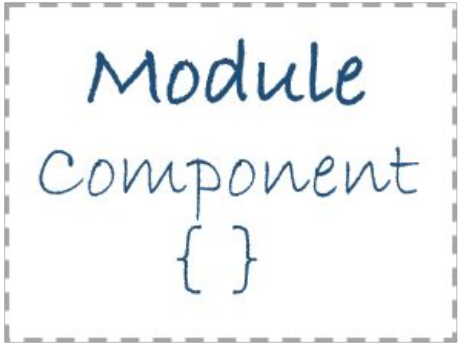
- The most common being **@NgModule, @Component, @Injectable, @Input, @Output**, etc.

- Each decorator accepts a well-defined, specific *configuration* object that contains information about the class or property to be decorated.

  Example :

  @NgModule decorator may define imports, exports, providers, definitions, etc,

Modules



Angular applications are composed of modules.

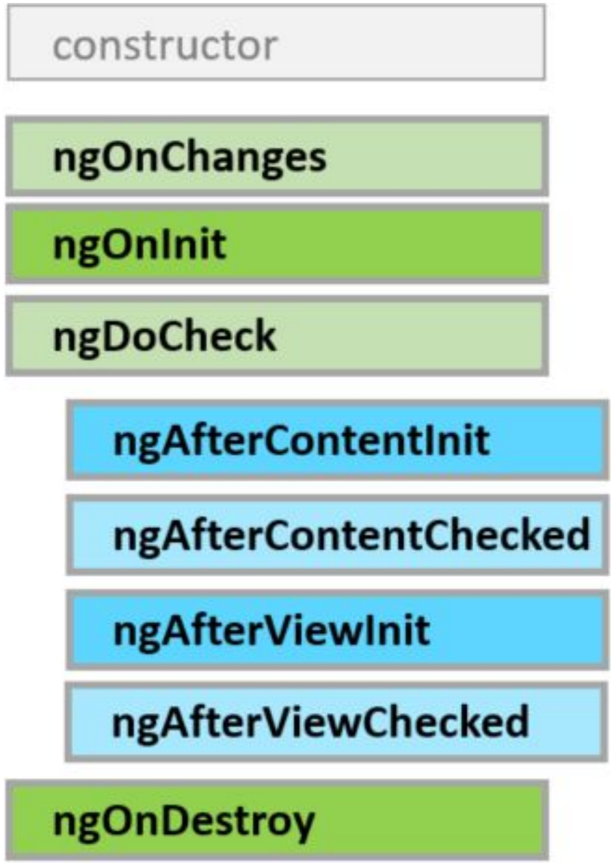
Every Angular app has *at least one Angular module class*, the root module, conventionally named **AppModule**.

Technically, Angular module is a class with an @NgModule decorator.
The most important properties are:

Declarations	the view classes that **belong to this module**
exports	the subset of declarations that should be visible and **usable in the other modules**
imports	other modules whose exported classes are **used in these module**
providers	creators of services that this module **contributes to the global collection of services**
bootstrap	the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

Intro to Lifecycle hooks

- Developers can tap into key moments in that lifecycle by implementing one or more of the lifecycle hook interfaces



Index

Interpolation, Pipes

What is INTERPOLATION:

mechanism to display a **component property in template**

PIPES:

```
{{ value }}

{{ hero.name | uppercase }}
```

```
<h2>My favorite hero is: {{myHero | uppercase}}</h2>
```

Directives

A directive is a custom HTML element that is used to extend the power of HTML.

- Angular transforms the DOM according to the instructions given by directives.
- A directive is a class with a **@Directive** decorator

Metadata
Directive
{ }

@Component decorator is actually a **@Directive** decorator extended with template-oriented features.

kinds of directives exist:
1 - A component is a directive-with-a-template
2 - Structural directives
3 - Attribute directives

Structural directives

Structural directives alter layout by adding, removing, and replacing elements in DOM.

```
<li *ngFor="let hero of heroes"></li>

<hero-detail *ngIf="selectedHero"></hero-detail>
```

*ngFor tells Angular to stamp out one <li> per hero in the heroes list.

*ngIf includes the HeroDetail component only if a selected hero exists.

Attribute directives

Attribute directives alter the appearance or behavior of an existing element.

`<input [(ngModel)]="hero.name">`

adds two-way data binding to an HTML form element.

few more directives that either alter the layout structure

for example, ngSwitch - *switch case*

or modify aspects of DOM elements and components

for example, ngStyle - *adds and removes a set of HTML inline styles.*

ngClass - *adds and removes a set of CSS classes.*

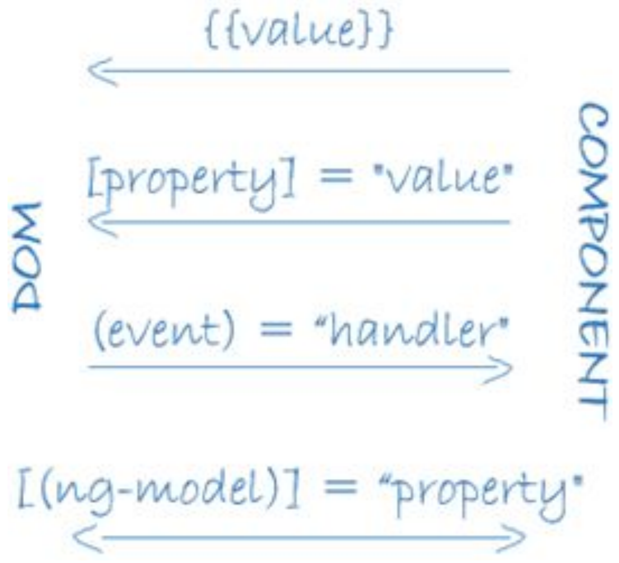
Of course, you can also write your own directives.

`ngModel needs importing FormsModule`

Data Binding

mechanism for coordinating parts of a template with parts of a component.

[property]	Data binding
e.g. `[hidden]=”true`	
[(ng-model)]	Two way data binding
e.g. `[(ngModel)]=”dataValue”`	
(event)	Event binding
e.g `(click)=”function”`	



Index

Forms

Data-bound user controls, tracks changes, validates input, and presents errors.

Types

1. Model Driven (Reactive Forms)
   a. **Reactive forms** are more robust: they're more scalable, reusable, and testable.
   b. If forms are a key part of your application, or you're already using reactive patterns for building your application.

2. Template Driven Forms
   a. **Template-driven forms** are useful for adding a simple form to an app, such as an email list signup form
   b. They're easy to add to an app, but they don't scale as well as reactive forms.

Forms Common foundation

- FormControl

  tracks the value and validation status of an **individual** form control.

- FormGroup

  tracks the same values and status for a **collection** of form controls.

- FormArray

  tracks the same values and status for an **array** of form controls.

Forms Validation

Improve data quality by validating user input for **accuracy** and **completeness**.

1. Template-driven validation
   a. Same validation attributes as you would do with native HTML form validation.
      E.g required, minlength
   b. Need to check **dirty** and **touched** properties

2. Reactive form validation
   a. You add validator functions **directly to the form control model in the component class**
   b. Built-in validators or Custom validators
   c. Style form control elements according to the state of the form.

      .ng-valid .ng-invalid .ng-dirty .ng-untouched .ng-touched etc

Index

Component Styles

Angular applications are styled with standard CSS.

That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.

Additionally, Angular can bundle **component styles** with components, enabling a more modular design than regular stylesheets.

Nested Components

Keeping all features in one component as the application grows will not be maintainable.

If you think of a typical web page we can normally break it down into a set of logical components each with its own view,

for example page can be broken up into a **header**, **footer** and perhaps a **sidebar**.

Split up large components into smaller sub-components, each focused on a **specific task** or **workflow.**

Interaction between Components

1. Pass data from parent to child with input binding
2. Parent listens for child event
3. Parent interacts with child via local variable
4. Parent and children communicate via a service

Index

Services

WHY services ?

A service is typically a class with a narrow, well-defined purpose.

It should do something specific and do it well.

Examples include:

- logging service
- data service
- application configuration

easy to factor your application logic into services and make those services available to components through dependency injection.

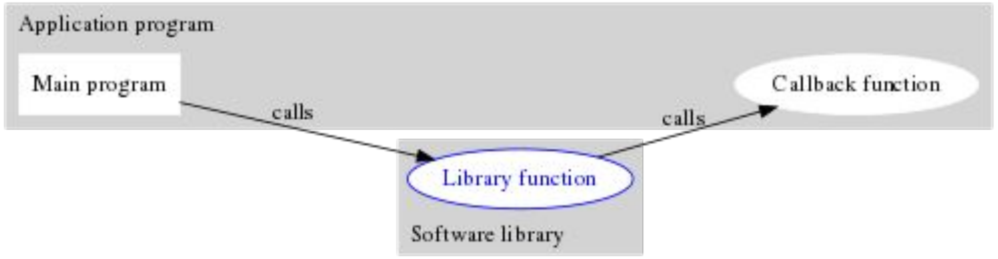
Observable data and Callbacks

The **Service** must wait for the **server to respond**,

getData() calls cannot return immediately with data, and the browser will not block while the service waits.

It can take a callback.

It could return a Promise.



It could return an Observable.

Promises

The Promise object represents
eventual completion
(or failure)
of an **asynchronous operation**, and its resulting value.

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then…
  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

```
promise.then(function(result) {
  console.log("Promise worked");
}, function(err) {
  console.log("Something broke");
});
```

Observable

An Observable is a unique Object **similar to a Promise** that can help manage async code.

Angular uses a third-party library called Reactive Extensions (RxJS).

Observables are used within Angular itself, including Angular’s event system and its **http client** service.

```
getHeroes(): Observable<Hero[]> {
  return of(HEROES);
}
```

```
getHeroes(): void {
  this.heroService.getHeroes()
      .subscribe( heroes =>
                this.heroes = heroes
        );
}
```

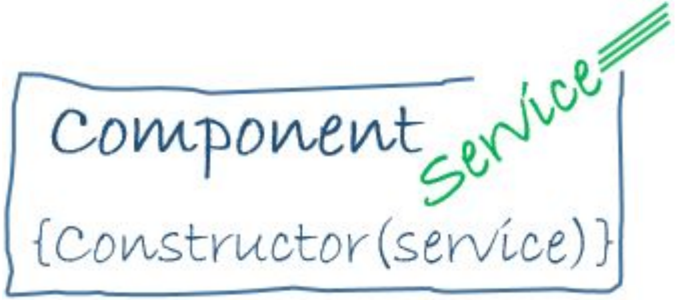
Dependency injection

Dependency injection is a way to **supply a new instance of a class** with the fully-formed dependencies it requires.

Most dependencies are services.

```
constructor(private service: HeroService) { }
```

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

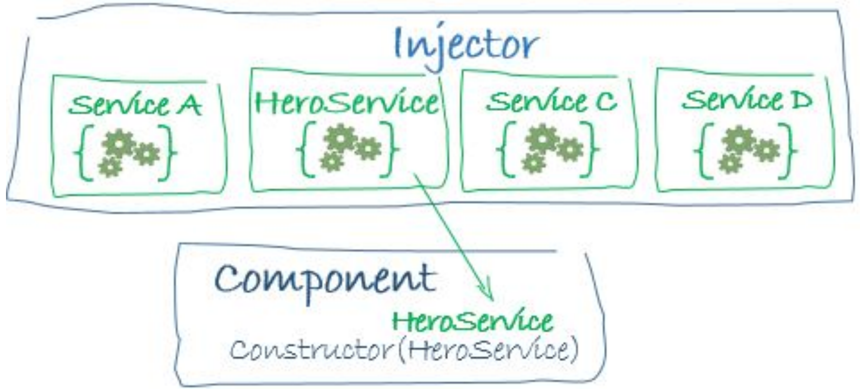


How dependency injection works

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

```
providers: [
  BackendService,
  HeroService,
  LoggerService
]

@Component({
  selector:    'hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
```



must have **previously registered a provider** of the Service with the injector.

You can register providers in modules or in components

Index

Retrieving data using HTTP

Example Covers Following :

- Angular HttpClient
- Getting JSON data
- Reading the full response
- Error handling
- Observables

Index

Navigation and Routing basics

Traditional Multi Page Application

- In a multi-page application, each page has its own, well defined url.
- Links to other pages are just anchor tags referencing the next landing page.

Single Page Application

- In SPA the page is just one, where all DOM elements get removed and created on the fly using JavaScript.

What is Router ?
The router is the framework that takes care of this, and makes a SPA behave like a MPA.

Navigation and Routing basics

The Angular Router **enables navigation** from one view to the next as users perform application tasks.

Whole process

1. The application has a configured **router**.
2. The shell component has a **RouterOutlet** where it can display views produced by the router.
3. It has **RouterLinks** that users can click to navigate via the router.

A routed Angular application has one singleton instance of the Router service. When the browser's URL changes, that router looks for a corresponding Route from which it can determine the component to display.

Navigation and Routing basics

Router outlet

The **RouterOutlet** is a *directive* from the router library that is used like a component.
It acts as a placeholder that marks the spot in the template where the router should display the components for that outlet.

```
<router-outlet></router-outlet>
```

Router links The RouterLink *directives* on the *anchor tags* give the router control over those elements.

```
<nav>
    <a routerLink="/crisis-center" routerLinkActive="active">Crisis Center</a>
    <a routerLink="/heroes" routerLinkActive="active">Heroes</a>
</nav>
```

Routing Implementation

Add the AppRoutingModule

```
ng generate module app-routing --flat --module=app
```

Add routes

RouterModule.forRoot()

Add RouterOutlet

Thank you