* A component controls a patch of screen called a **view**.
* For example, the following views are controlled by components:
* The **app root** with the **navigation** **links**.
* 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.

export class HeroListComponent implements OnInit {

heroes: Hero[];

selectedHero: Hero;

constructor(private service: HeroService) { }

ngOnInit() {

this.heroes = this.service.getHeroes();

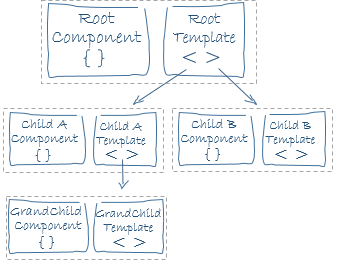
}

selectHero(hero: Hero) { this.selectedHero = hero; }

}

* 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](https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html), like ngOnInit() declared above.

**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. Here is a template for our HeroListComponent:
* Notice how <hero-detail> rests comfortably among native HTML elements. Custom components mix seamlessly with native HTML in the same layouts.

**Metadata**

* Metadata tells Angular how to process a class.
* [Looking back at the code](https://angular.io/docs/ts/latest/guide/architecture.html#component-code) for HeroListComponent, you can see that it's just a class. There is no evidence of a framework, no "Angular" in it at all.
* In fact, HeroListComponent really is *just a class*. It's not a component until you *tell Angular about it*.
* To tell Angular that HeroListComponent is a component, attach **metadata** to the class.

@Component({

moduleId: module.id,

selector: 'hero-list',

templateUrl: 'hero-list.component.html',

providers: [ HeroService ]

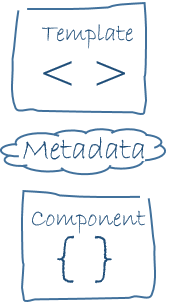
})

export class HeroListComponent implements OnInit {

/\* . . . \*/

}

* 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 possible @Component configuration options:
  + **moduleId**: sets the source of the base address (module.id) for module-relative URLs such as the templateUrl.
  + **selector**: CSS selector that tells Angular to create and insert an instance of this component where it finds a <hero-list> tag in parent HTML. For example, if an app's HTML contains <hero-list></hero-list>, then Angular inserts an instance of the HeroListComponent view between those tags.
  + **templateUrl**: module-relative address of this component's HTML template, shown [above](https://angular.io/docs/ts/latest/guide/architecture.html#templates).
  + **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.

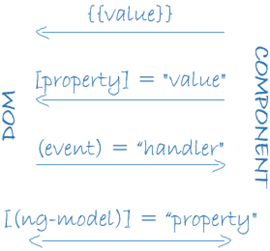
The metadata in the @Component tells Angular where to get the major building blocks you specify for the component.

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.

## 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.

The HeroListComponent [example](https://angular.io/docs/ts/latest/guide/architecture.html#templates) template has three forms:

#### app/hero-list.component.html (binding)

<li>{{hero.name}}</li>

<hero-detail [hero]="selectedHero"></hero-detail>

<li (click)="selectHero(hero)"></li>

* The {{hero.name}} [interpolation](https://angular.io/docs/ts/latest/guide/displaying-data.html#interpolation) displays the component's hero.name property value within the <li> element.
* The [hero] [property binding](https://angular.io/docs/ts/latest/guide/template-syntax.html#property-binding) passes the value of selectedHero from the parent HeroListComponent to the hero property of the child HeroDetailComponent.
* The (click) [event binding](https://angular.io/docs/ts/latest/guide/user-input.html#click) calls the component's selectHero method when the user clicks a hero's name.

**Two-way data binding** is an important fourth form that combines property and event binding in a single notation, using the ngModel directive. Here's an example from the HeroDetailComponent template:

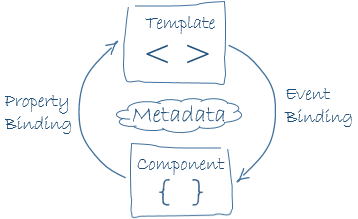
#### app/hero-detail.component.html (ngModel)

COPY CODE

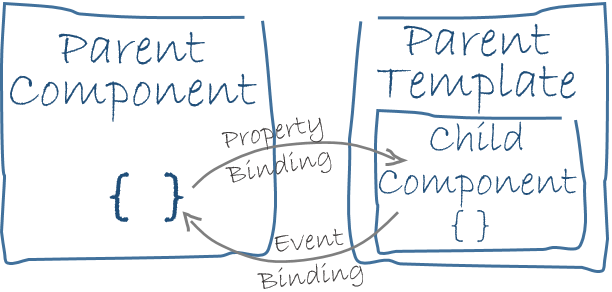
<input [(ngModel)]="hero.name">

In two-way binding, a data property value flows to the input box from the component as with property binding. The user's changes also flow back to the component, resetting the property to the latest value, as with event binding.

Angular processes all data bindings once per JavaScript event cycle, from the root of the application component tree through all child components.



Data binding plays an important role in communication between a template and its component.



Data binding is also important for communication between parent and child components.