Introduction to components

A *component* controls a patch of screen called a view. For example, individual components define and control each of the following views from the Tutorial:

- The app root with the navigation links.
- The list of heroes.
- The hero editor.

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.

For example, `HeroListComponent` has a `heroes` property that holds an array of heroes. Its `selectHero()` method sets a `selectedHero` property when the user clicks to choose a hero from that list. The component acquires the heroes from a service, which is a TypeScript parameter property on the constructor. The service is provided to the component through the dependency injection system.

src/app/hero-list.component.ts (class)

```
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, like `ngOnInit()`.

Component metadata

The `@Component` decorator identifies the class immediately below it as a component class, and specifies its metadata. In the example code below, you can see that `HeroListComponent` is just a class, with no special Angular notation or syntax at all. It's not a component until you mark it as one with the `@Component` decorator.

The metadata for a component tells Angular where to get the major building blocks that it needs to create and present the component and its view. In particular, it associates a *template* with the component, either directly with inline code, or by reference. Together, the component and its template describe a view.

In addition to containing or pointing to the template, the `@Component` metadata configures, for example, how the component can be referenced in HTML and what services it requires.

Here's an example of basic metadata for `HeroListComponent`.

src/app/hero-list.component.ts (metadata)

```
@Component({
  selector:    'app-hero-list',
  templateUrl: './hero-list.component.html',
  providers:  [ HeroService ]
})
export class HeroListComponent implements OnInit {
  /* . . . */
}
```

This example shows some of the most useful `@Component` configuration options:

- `selector`: A CSS selector that tells Angular to create and insert an instance of this component wherever it finds the corresponding tag in template HTML. For example, if an app's HTML contains `<app-hero-list></app-hero-list>`, then Angular inserts an instance of the `HeroListComponent` view between those tags.
- `templateUrl`: The module-relative address of this component's HTML template. Alternatively, you can provide the HTML template inline, as the value of the `template` property. This template defines the component's *host view*.
- `providers`: An array of providers for services that the component requires. In the example, this tells Angular how to provide the `HeroService` instance that the component's constructor uses to get the list of heroes to display.

Templates and views

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.

Views are typically arranged hierarchically, allowing you to modify or show and hide entire UI sections or pages as a unit. The template immediately associated with a component defines that component's *host view*. The component can also define a view *hierarchy*, which contains *embedded views*, hosted by other components.

A view hierarchy can include views from components in the same NgModule, but it also can (and often does) include views from components that are defined in different NgModules.

Template syntax

A template looks like regular HTML, except that it also contains Angular template syntax, which alters the HTML based on your app's logic and the state of app and DOM data. Your template can use *data binding* to coordinate the app and DOM data, *pipes* to transform data before it is displayed, and *directives* to apply app logic to what gets displayed.

For example, here is a template for the Tutorial's `HeroListComponent`.

src/app/hero-list.component.html

```
<h2>Hero List</h2>

<p><i>Pick a hero from the list</i></p>
<ul>
  <li *ngFor="let hero of heroes" (click)="selectHero(hero)">
    {{hero.name}}
  </li>
</ul>

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

This template uses typical HTML elements like `<h2>` and `<p>`, and also includes Angular template-syntax elements, `*ngFor`, `{{hero.name}}`, `(click)`, `[hero]`, and `<app-hero-detail>`. The template-syntax elements tell Angular how to render the HTML to the screen, using program logic and data.

- The `*ngFor` directive tells Angular to iterate over a list.
- `{{hero.name}}`, `(click)`, and `[hero]` bind program data to and from the DOM, responding to user input. See more about data binding below.
- The `<app-hero-detail>` tag in the example is an element that represents a new component, `HeroDetailComponent`. `HeroDetailComponent` (code not shown) defines the hero-detail child view of `HeroListComponent`. Notice how custom components like this one seamlessly with native HTML in the same layouts.

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 and pull logic by hand is tedious, error-prone, and a nightmare to read, as any experienced front-end JavaScript programmer can attest.

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

The following diagram shows the four forms of data binding markup. Each form has a direction: to the DOM, from the DOM, or both.

This example from the `HeroListComponent` template uses three of these forms.

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

```
<li>{{hero.name}}</li>
<app-hero-detail [hero]="selectedHero"></app-hero-detail>
<li (click)="selectHero(hero)"></li>
```

- The `{{hero.name}}` *interpolation* displays the component's `hero.name` property value within the `<li>` element.
- The `[hero]` *property binding* passes the value of `selectedHero` from the parent `HeroListComponent` to the `hero` property of the child `HeroDetailComponent`.
- The `(click)` *event binding* calls the component's `selectHero` method when the user clicks a hero's name.

Two-way data binding (used mainly in template-driven forms) combines property and event binding in a single notation. Here's an example from the `HeroDetailComponent` template that uses two-way data binding with the `ngModel` directive.

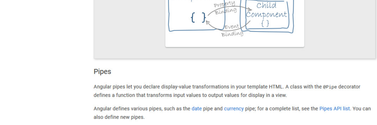
src/app/hero-detail.component.html (ngModel)

```
<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 for each 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, and is also important for communication between parent and child components.



Pipes

Angular pipes let you declare display-value transformations in your template HTML. A class with the `@Pipe` decorator defines a function that transforms input values to output values for display in a view.

Angular defines various pipes, such as the date pipe and currency pipe; for a complete list, see the Pipes API list. You can also define new pipes.

To specify a value transformation in an HTML template, use the pipe operator (|).

```
{{interpolated_value | pipe_name}}
```

You can chain pipes, sending the output of one pipe function to be transformed by another pipe function. A pipe can also take arguments that control how it performs its transformation. For example, you can pass the desired format to the `date` pipe.

```
<!-- Default format: output 'Jun 15, 2015'-->
<p>Today is {{today | date}}</p>

<!-- fullDate format: output 'Monday, June 15, 2015'-->
<p>The date is {{today | date:'fullDate'}}</p>

<!-- shortTime format: output '9:43 AM'-->
<p>The time is {{today | date:'shortTime'}}</p>
```

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 technically a directive. However, components are so distinctive and central to Angular applications that Angular defines the `@Component()` decorator, which extends the `@Directive()` decorator with template-oriented features.

In addition to components, there are two other kinds of directives: *structural* and *attribute*. Angular defines a number of directives of both kinds, and you can define your own using the `@Directive()` decorator.

Just as for components, the metadata for a directive associates the decorated class with a `selector` element that you use to insert it into HTML. In templates, directives typically appear within an element tag as attributes, either by name or as the target of an assignment or a binding.

Structural directives

Structural directives alter layout by adding, removing, and replacing elements in the DOM. The example template uses two built-in structural directives to add application logic to how the view is rendered.

src/app/hero-list.component.html (structural)

```
<li *ngFor="let hero of heroes"></li>
<app-hero-detail *ngIf="selectedHero"></app-hero-detail>
```

- `*ngFor` is an iterative; it tells Angular to stamp out one `<li>` per hero in the `heroes` list.
- `*ngIf` is a conditional; it includes the `HeroDetail` component only if a selected hero exists.

Attribute directives

Attribute directives alter the appearance or behavior of an existing element. In templates they look like regular HTML attributes, hence the name.

The `ngModel` directive, which implements two-way data binding, is an example of an attribute directive. `ngModel` modifies the behavior of an existing element (typically `<input>`) by setting its display value property and responding to change events.

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

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

Angular has more pre-defined directives that either alter the layout structure (for example, ngSwitch) or modify aspects of DOM elements and components (for example, ngStyle and ngClass).

> Learn more in the Attribute Directives and Structural Directives guides.