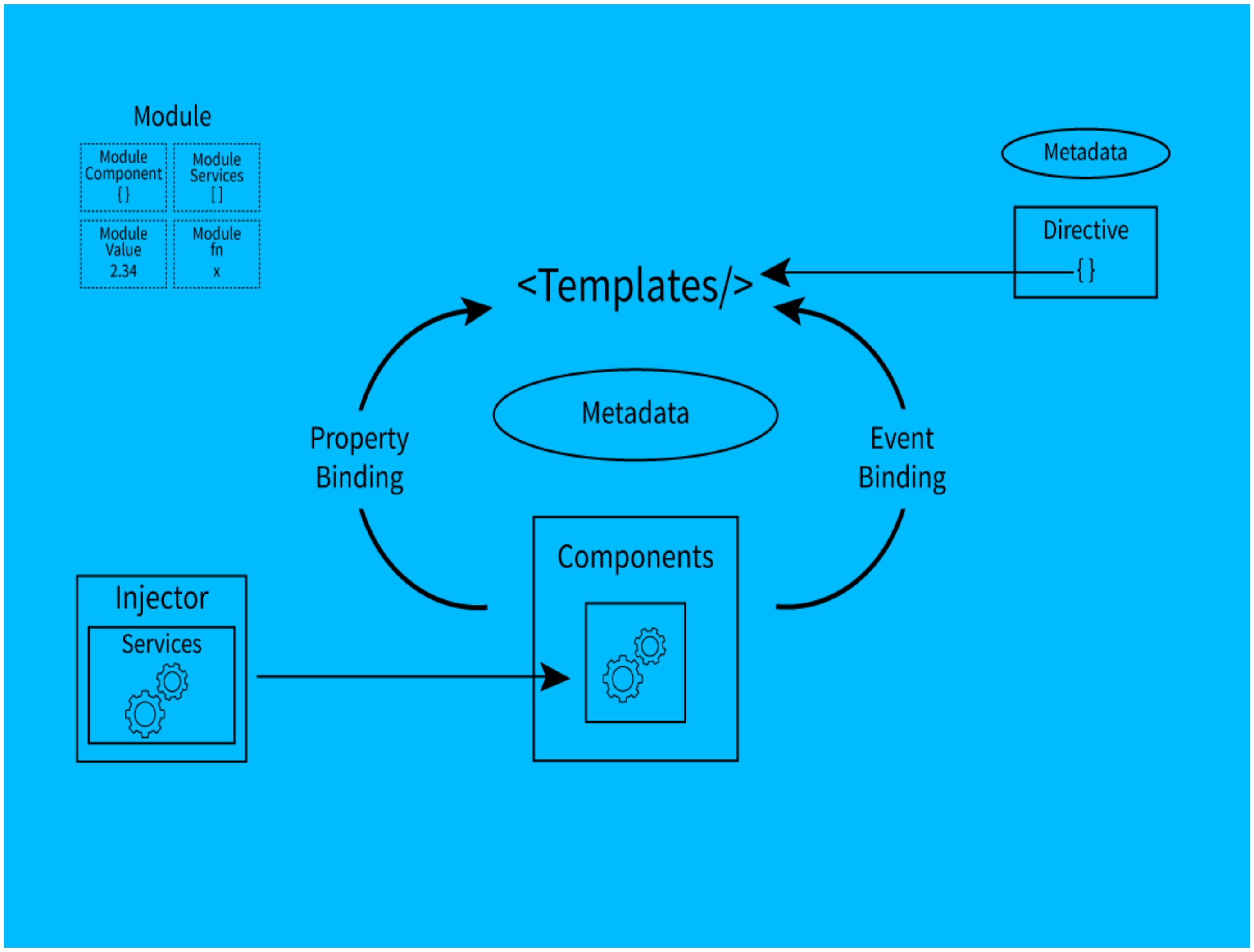
There are basically 8 building blocks of Angular 2. These are:

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

Let's go over each one of them one by one.



Modules

Modules are blocks of code that do a certain type of task. A module exports some value in the main code, such as class. The first and the most common module that you will study is the one that exports *Component* Class.

app/app.component.ts (excerpt)

```
export class AppComponent { }
```

Here **app.component** is a module.

Libraries

Libraries' names start with the *@angular* prefix. Modules can be a library of other modules. Angular 2 itself has many modules that are libraries of others.

@angular/core is the most important library that contains most of the modules that we need.

Components

A component is basically a class that is used to show an element on the screen. The components have some properties and by using them we can manipulate how the element should look and behave on the screen. We can create a component, destroy, and update as the user moves in the application. Life Cycle hooks are the modules that we use for this purpose. Like `ngOnInit()`

app/app.component.ts (excerpt)

```
export class blogComponent { }
```

Here **blogComponent** is a component.

Templates

The view of the component is defined through templates. Templates are basically the HTML we use to show on our page.

app/hero-list.component.html

```
   <h2>Hero List</h2>
   <p><i>Pick a hero from the list</i></p>
   <ul>
    <li *ngFor="let food of foods" (click)="selectedFood(food)">
      {{food.name}}
    </li>
   </ul>
<food-detail *ngIf="selectedFood" [food]="selectedHero"></food-detail>
```

This is a simple HTML file, but you may wonder: *What are these elements?*

<Is there an answer to this question?>
***ngFor,{{food.name}}, (click), [food], and <food-detail>?**

Metadata

Metadata tells Angular how a class should be processed on the screen. For example:

```
@Component({
 selector: 'food-list',
 templateUrl: 'app/food-list.component.html',
 directives:  [FoodDetailComponent],
 providers:   [FoodService]
})
```

To tell Angular that we are using a component with certain metadata, we attach a decorator to it ("@").

Here **@Component** will be identified as a component class.

Selector tells Angular to render the HTML that templateURL has at this tag <food-list></food-list>

Directives are other components that this Component will require to render and providers are the services required.

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

Data Binding

The main feature of any JavaScript framework is data binding. As Angular 1, Angular 2 also support data binding.

There are 4 ways of binding a data according to the direction to the DOM, from the DOM, or in both directions:

```
<input [(ngModel)]="food.name">
<li>{{food.name}}</li>
<food-detail [food]="selectedFood"></food-detail>
<li (click)="selectFood(food)"></li>
```

1. The {{hero.name}} *interpolation* display food.name value in li tag.
2. The [hero] *property binding* passes the selected food value from parent to child component.
3. The (click) *event binding* calls the selectedFood function when a user clicks on it.
4. Two-way data binding is an important fourth way that combines property and event binding by the ngModel directive.

Directive

Directive helps us to add behavior to the DOM elements. We can attach multiple directives to the DOM elements. In TypeScript we define decoratives by @*decorative* decorator.

There are 3 types of decorative:

1. Directive-with-a-template
2. Structural
3. Attribute

A component is a *directive-with-a-template*:

Structural directives add, delete and replace DOM elements. For example:

```
<li *ngFor = "let food of foods”></li>
<food-detail *ngIf="selectedFood"></food-detail>
```

Attribute directives change the appearance of DOM elements. For example:

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

Services

A service is a class containing any function, feature with a defined, and specific purpose. For example:

```
export class FoodService {
 getfoodies(): Food[] {
   return FOODIES;
 }
}
```

Dependency Injections

Dependency injection allows one to inject a dependency as a service throughout the web application. To inject a dependency we do not create a service but we have a constructor to request the service. The framework then provides it. For example:

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