ARCHITECTURE OVERVIEW

Developer Preview Only - some details may change

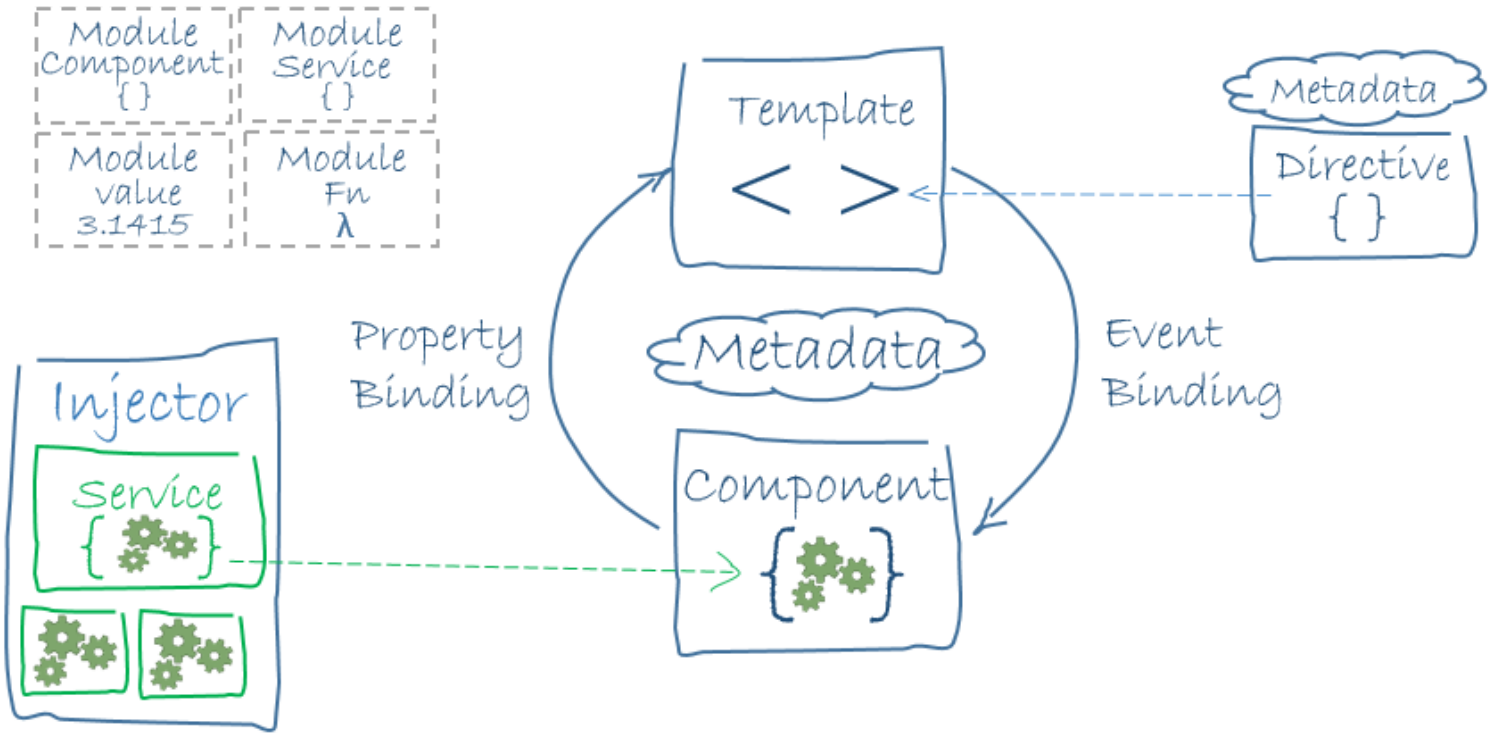
Angular 2 is a framework to help us build client applications in HTML and JavaScript.

The framework consists of several cooperating libraries, some of them core and some optional.

We write applications by composing HTML *templates* with Angularized-markup, writing *component* classes to manage those templates, adding application logic in *services*, and handing the top root component to Angular's *bootstrapper*.

Angular takes over, presenting our application content in a browser and responding to user interactions according to the instructions we provided.

Of course there is more to it than this. We'll learn the details when we dive into the guide chapters. Let's get the big picture first.



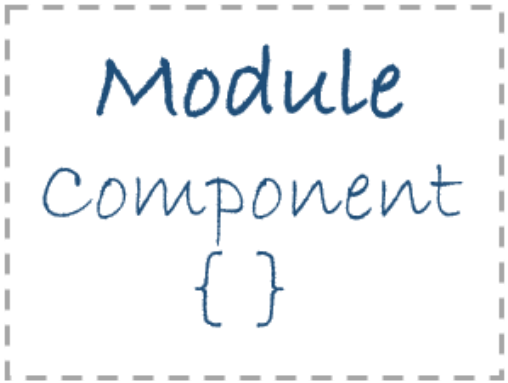
The architecture diagram identifies the eight main building blocks of an Angular 2 application:

1. Module
2. Component
3. Template
4. Metadata
5. Data Binding
6. Service
7. Directive
8. Dependency Injection

Learn these eight and we're on our way.

> The code referenced in this chapter is available as a live example.

The Module



Angular apps are modular.

In general we assemble our application from many **modules**.

A typical module is a cohesive block of code dedicated to a single purpose. A module **exports** something of value in that code, typically one thing such as a class.

> **Modules are optional**
>
> We highly recommend modular design. TypeScript has great support for ES2015 module syntax and our chapters assume we're taking a modular approach using that syntax. That's why we list *Module* among the basic building blocks.
>
> Angular itself doesn't require a modular approach nor this particular syntax. Don't use it if you don't want it. Each chapter has plenty to offer after you steer clear of the `import` and `export` statements.
>
> Find setup and organization clues in the JavaScript track (select it from the combobox at the top of this page) which demonstrates

> Angular 2 development with plain old JavaScript and no module system.

Perhaps the first module we meet is a module that exports a *component* class. The component is one of the basic Angular blocks, we write a lot of them, and we'll talk about components in the next segment. For the moment it is enough to know that a component class is the kind of thing we'd export from a module.

Most applications have an `AppComponent`. By convention, we'll find it in a file named `app.component.ts`. Look inside such a file and we'll see an `export` statement like this one.

app/app.component.ts (excerpt)

```
export class AppComponent { }
```

The `export` statement tells TypeScript that this is a module whose `AppComponent` class is public and accessible to other modules of the application.

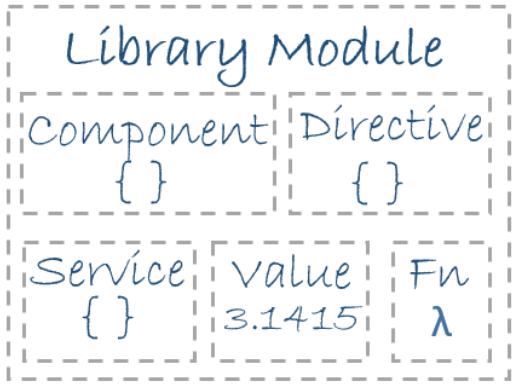
When we need a reference to the `AppComponent`, we **import** it like this:

app/main.ts (excerpt)

```
import {AppComponent} from './app.component';
```

The `import` statement tells the system it can get an `AppComponent` from a module named `app.component` located in a neighboring file. The **module name** (AKA module id) is often the same as the filename without its extension.

Library Modules



Some modules are libraries of other modules.

Angular itself ships as a collection of library modules called "barrels". Each Angular library is actually a public facade over several logically related private modules.

The `angular2/core` library is the primary Angular library module from which we get most of what we need.

There are other important Angular library modules too such as `angular2/common`, `angular2/router`, and `angular2/http`.

> Learn more about how Angular organizes and distributes modules in "Modules, barrels and bundles".

We import what we need from an Angular library module in much the same way. For example, we import the Angular `Component` ***function*** from the *angular2/core* module like this:

```
import {Component} from 'angular2/core';
```

Compare that syntax to our previous import of `AppComponent`.

```
import {AppComponent} from './app.component';
```

Notice the difference? In the first case, when importing from an Angular library module, the import statement refers to the bare module name, `angular2/core`, *without a path prefix*.

When we import from one of *our* own files, we prefix the module name with the file path. In this example we specify a relative file path (./). That means the source module is in the same folder (./) as the module importing it. We could path up and around the application folder structure if the source module were somewhere else.

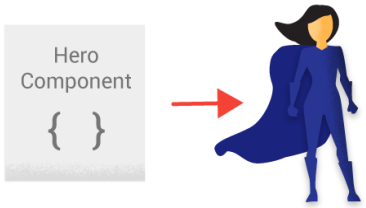
> We import and export in the ECMAScript 2015 (ES2015) module syntax. Learn more about that syntax here and many other places on the web.
>
> The infrastructure *behind* module loading and importing is an important subject. But it's a subject outside the scope of this introduction to Angular. While we're focused on our application, *import* and *export* is about all we need to know.

The key take aways are:

- Angular apps are composed of modules.
- Modules export things — classes, function, values — that other modules import.
- We prefer to write our application as a collection of modules, each module exporting one thing.

The first module we write will most likely export a component.

The Component



A **Component** controls a patch of screen real estate that we could call a *view*. The shell at the application root with navigation links, that list of heroes, the hero editor ... they're all views controlled by Components.

We 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.

A `HeroListComponent`, for example, might have a `heroes` property that returns an array of heroes that it acquired from a service. It might have a `selectHero()` method that sets a `selectedHero` property when the user click on a hero from that list. It might be a class like this:

app/hero-list.component.ts

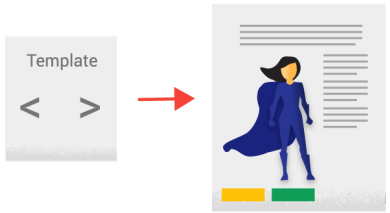
```
export class HeroListComponent implements OnInit {
  constructor(private _service: HeroService){ }

  heroes:Hero[];
  selectedHero: Hero;

  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. The developer can take

action at each moment in this lifecycle through optional Lifecycle Hooks.

> We're not showing those hooks in this example but we are making a mental note to find out about them later.
>
> We may wonder who is calling that constructor? Who provides the service parameter? For the moment, have faith that Angular will call the constructor and deliver an appropriate `HeroService` when we need it.

The Template



We 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 much of the time ... and then it gets a bit strange. Here is a template for our `HeroList` component.

app/hero-list.component.html

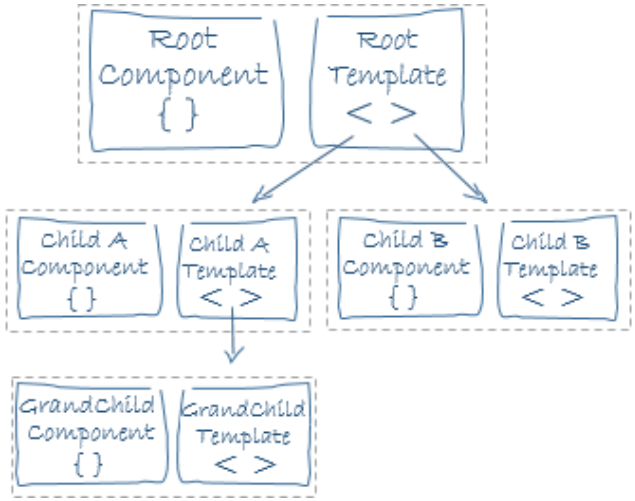
```
<h2>Hero List</h2>

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

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

We recognize `<h2>` and `<div>`. But there's other markup that no one told us about in school. What are `*ngFor`, `{{hero.name}}`, `(click)`, `[hero]`, and `<hero-detail>`?

These are examples of Angular's template syntax. We will grow accustomed to that syntax and may even learn to love it. We'll begin to explain it in a moment.

Before we do, focus attention on the last line. The `<hero-detail>` tag is a custom element representing the `HeroDetailComponent`.

The `HeroDetailComponent` is a *different* component than the `HeroListComponent` we've been reviewing. The `HeroDetailComponent` (code not shown) presents facts about a particular hero, the hero that the user selects from the list presented by the the `HeroListComponent`. The `HeroDetailComponent` is a **child** of the the `HeroListComponent`.



Notice how `<hero-detail>` rests comfortably among the HTML elements we already know. We can mix ... and will mix ... our custom components with native HTML in the same layouts.

And in this manner we can and will compose complex component trees to build out our richly featured application.

Angular Metadata

Metadata tells Angular how to process a class.

Looking back at the `HeroListComponent`, we see that it's just a class. There is no evidence of a framework, no "Angular" in it at all.

In fact, it really is *just a class*. It's not a component until we *tell Angular about it*.

We tell Angular that `HeroListComponent` is a component by attaching **metadata** to the class.

The easy way to attach metadata in TypeScript is with a **decorator**. Here's some metadata for `HeroListComponent`:

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

```
1. @Component({
2.   selector:    'hero-list',
3.   templateUrl: 'app/hero-list.component.html',
4.   directives:  [HeroDetailComponent],
5.   providers:   [HeroService]
6. })
7. export class HeroesComponent { ... }
```

Here we see the `@Component` decorator which (no surprise) identifies the class immediately below it as a Component class.

A decorator is a function. Decorators often have a configuration parameter. The `@Component` decorator takes a required configuration object with the information Angular needs to create and present the component and its view.

Here we see a few of the possible `@Component` configuration options:

- `selector` - a css selector that tells Angular to create and insert an instance of this component where it finds a `<hero-list>` tag in *parent* HTML. If the template of the application shell (a Component) contained

```
<hero-list></hero-list>
```

  Angular inserts an instance of the `HeroListComponent` view between those tags.

- `templateUrl` - the address of this component's template which we showed above.

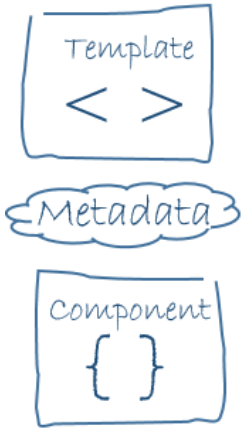
- `directives` - an array of the Components or Directives that *this* template requires. We saw in the last line of our template that we expect Angular to insert a `HeroDetailComponent` in the space indicated by `<hero-detail>` tags. Angular will do so only if we mention the `HeroDetailComponent` in this `directives` array.

- `providers` - an array of **dependency injection providers** for services that the component requires. This is one way to tell Angular that our component's constructor requires a `HeroService` so it can get the list of heroes to display. We'll get to dependency injection in a moment.

The `@Component` function takes the configuration object and turns it into metadata that it attaches to the component class definition. Angular discovers this metadata at runtime and thus knows how to do "the right thing".

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

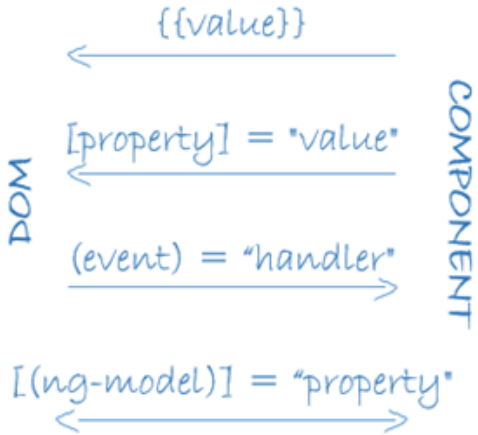
We apply other metadata decorators in a similar fashion to guide Angular behavior. The `@Injectable`, `@Input`, `@Output`, `@RouterConfig` are a few of the more popular decorators we'll master as our Angular knowledge grows.



The architectural take-away is that we must add metadata to our code so that Angular knows what to do.

Data Binding

Without a framework, we 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 the experienced jQuery programmer can attest.



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

There are four forms of data binding syntax. Each form has a direction - to the DOM, from the DOM, or in both directions - as indicated by the arrows in the diagram.

We saw three forms of data binding in our example template:

app/hero-list.component.html (excerpt)

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

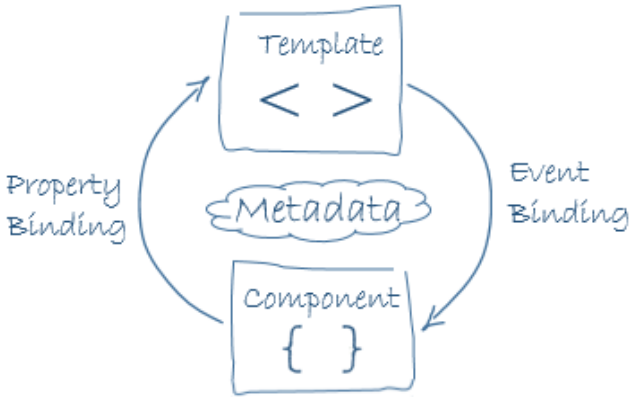
- The "interpolation" displays the component's `hero.name` property value within the `<div>` tags.
- The `[hero]` property binding passes the `selectedHero` from the parent `HeroListComponent` to the `hero` property of the child `HeroDetailComponent`.
- The `(click)` event binding calls the Component's `selectHero` method when the user clicks on 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. We didn't have a two-way binding in the `HeroListComponent` template; here's an example from the `HeroDetailComponent` template (not shown):

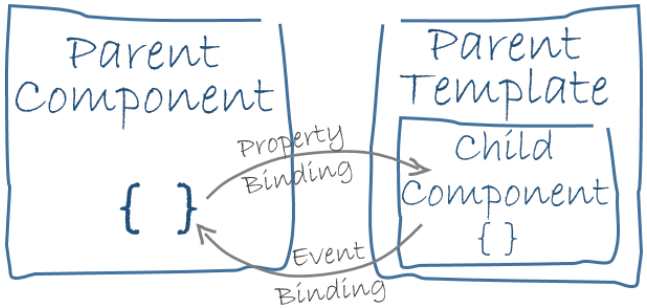
```
<input [(ngModel)]="hero.name">
```

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, depth-first from the root of the application component tree.



We don't know all the details yet but it's clear from these examples that data binding plays an important role in communication between a template and its component ...



... ***and*** between parent and child components

The Directive

Our Angular templates are *dynamic*. When Angular renders them, it transforms the DOM according to the instructions given by a **directive**.

A directive is a class with directive metadata. In TypeScript we'd apply the `@Directive` decorator to attach metadata to the class.

We already met one form of directive: the component. A component is a *directive-with-a-template* and the `@Component`

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

> While the **component is technically a directive**, it is so distinctive and central to Angular applications that we chose to separate the component from the directive in our architectural overview.

There are two *other* kinds of directives as well that we call "structural" and "attribute" directives.

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

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

We see two built-in structural directives at play in our example template:

```
<div *ngFor="#hero of heroes"></div>
<hero-detail *ngIf="selectedHero"></hero-detail>
```

- `*ngFor` tells Angular to stamp out one `<div>` per hero in the `heroes` list.
- `*ngIf` includes the `HeroDetail` component only if a selected hero exists.

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.

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

It modifies the behavior of an existing element (typically an `<input>`) by setting its display value property and responding to change events.

Angular ships with a small number of other directives that either alter the layout structure (e.g. ngSwitch) or modify aspects of DOM elements and components (e.g. ngStyle and ngClass).

And of course we can write our own directives.

The Service

"Service" is a broad category encompassing any value, function or feature that our 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.

Examples include:

- logging service
- data service
- message bus
- tax calculator

- application configuration

There is nothing specifically *Angular* about services. Angular itself has no definition of a *service*. There is no *ServiceBase* class.

Yet services are fundamental to any Angular application.

Here's an example of a service class that logs to the browser console

app/logger.service.ts (class only)

```
export class Logger {
  log(msg: any)   { console.log(msg); }
  error(msg: any) { console.error(msg); }
  warn(msg: any)  { console.warn(msg); }
}
```

Here's a `HeroService` that fetches heroes and returns them in a resolved promise. The `HeroService` depends on the `LoggerService` and another `BackendService` that handles the server communication grunt work.

app/hero.service.ts (class only)

```
export class HeroService {
  constructor(
    private _backend: BackendService,
    private _logger: Logger) { }
```

```
    private _heroes:Hero[] = [];

    getHeroes() {
      this._backend.getAll(Hero).then( (heroes:Hero[]) => {
        this._logger.log(`Fetched ${heroes.length} heroes.`);
        this._heroes.push(...heroes); // fill cache
      });
      return this._heroes;
    }
  }
```

Services are everywhere.

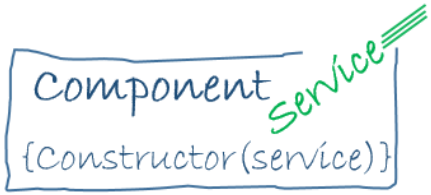
Our components are big consumers of services. They depend upon services to handle most chores. They don't fetch data from the server, they don't validate user input, they don't log directly to the console. They delegate such tasks to services.

A component's job is to enable the user experience and nothing more. It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a "model"). A good component presents properties and methods for data binding. It delegates everything non-trivial to services.

Angular doesn't *enforce* these principles. It won't complain if we write a "kitchen sink" component with 3000 lines.

Angular does help us *follow* these principles ... by making it easy to factor our application logic into services and make those services available to components through *dependency injection*.

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.

In TypeScript, Angular can tell which services a component needs by looking at the types of its constructor parameters. For example, the constructor of our `HeroListComponent` needs the `HeroService`:

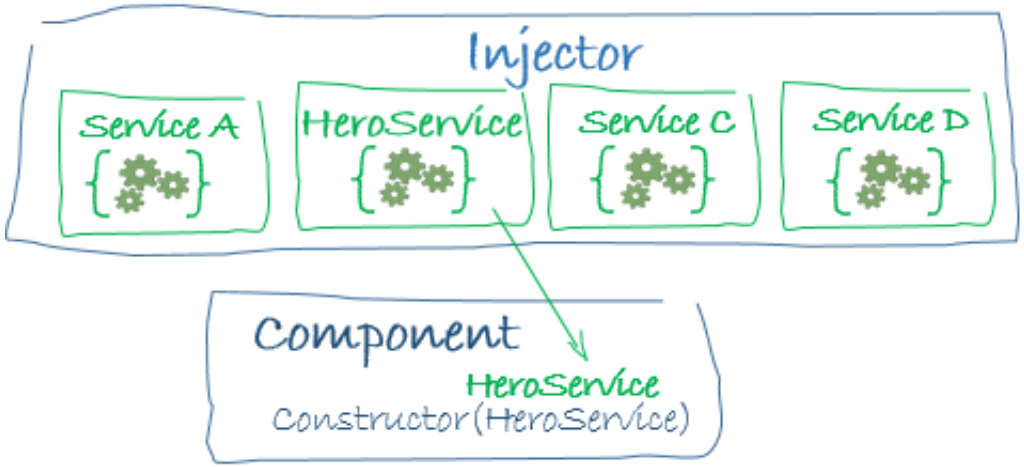
app/hero-list.component (constructor)

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

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 what we mean by *dependency injection*.

The process of `HeroService` injection looks a bit like this:



If the `Injector` doesn't have a `HeroService`, how does it know how to make one?

In brief, we must have previously registered a **provider** of the `HeroService` with the `Injector`. A provider is something that can create or return a service, typically the service class itself.

We can register providers at any level of the application component tree. We often do so at the root when we bootstrap the application so that the same instance of a service is available everywhere.

app/main.ts (excerpt)

```
bootstrap(AppComponent, [BackendService, HeroService, Logger]);
```

Alternatively, we might register at a component level ...

app/hero-list.component.ts (excerpt)

```
@Component({
  providers:   [HeroService]
```

```
})
export class HeroesComponent { ... }
```

... in which case we get a new instance of the service with each new instance of that component.

We've vastly over-simplified dependency injection for this overview. We can learn the full story in the Dependency Injection chapter.

The points to remember are:

- dependency injection is wired into the 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 using a *provider*.
- a *provider* is a recipe for creating a service.
- we register *providers* with injectors.

Wrap up

We've learned just a bit about the eight main building blocks of an Angular application

1. Module

2. Component
3. Template
4. Metadata
5. Data Binding
6. Service
7. Directive
8. Dependency Injection

That's a foundation for everything else in an Angular application and it's more than enough to get going. But it doesn't include everything we'll need or want to know.

The Other Stuff

Here is a brief, alphabetical list of other important Angular features and services. Most of them are covered in this Developers Guide (or soon will be):

Animations - A forthcoming animation library makes it easy for developers to animate component behavior without deep knowledge of animation techniques or css.

Bootstrap - A method to configure and launch the root application component.

Change Detection - Learn how Angular decides that a component property value has changed and when to update the screen. Learn how it uses **zones** to intercept asynchronous activity and run its change detection strategies.

Component Router - With the Component Router service, users can navigate a multi-screen application in a familiar web browsing style using URLs.

Events - The DOM raises events. So can components and services. Angular offers mechanisms for publishing and subscribing to events including an implementation of the RxJS Observable proposal.

Forms - Support complex data entry scenarios with HTML-based validation and dirty checking.

HTTP - Communicate with a server to get data, save data, and invoke server-side actions with this Angular HTTP client.

Lifecycle Hooks - We can tap into key moments in the lifetime of a component, from its creation to its destruction, by implementing the "Lifecycle Hook" interfaces.

Pipes - Services that transform values for display. We can put pipes in our templates to improve the user experience. For example, this `currency` pipe expression,

```
price | currency:'USD':true
```

displays a price of "42.33" as `$42.33`.

Testing - Angular provides a testing library for "unit testing" our application parts as they interact with the Angular framework.

> **Next Step**
>
> Displaying Data