[Building Blocks 3](#_Toc153732364)

[Directive 3](#_Toc153732365)

[Template, Component, Data Binding 3](#_Toc153732366)

[Service 4](#_Toc153732367)

[Metadata, Dependency Injection 5](#_Toc153732368)

[Angular Module 5](#_Toc153732369)

[Template Syntax 6](#_Toc153732370)

[HTML 6](#_Toc153732371)

[Template expression and statement 6](#_Toc153732372)

[Template Expression 6](#_Toc153732373)

[Template Statement 7](#_Toc153732374)

[Safe navigation operator 8](#_Toc153732375)

[Pipe 8](#_Toc153732376)

[Template Reference Variable 8](#_Toc153732377)

[Data Binding 9](#_Toc153732378)

[Data Binding Overview 9](#_Toc153732379)

[Property binding 11](#_Toc153732380)

[Attribute binding 11](#_Toc153732381)

[Class binding 11](#_Toc153732382)

[Style binding 11](#_Toc153732383)

[Event binding 12](#_Toc153732384)

[Two-way binding (Property and Event binding) 12](#_Toc153732385)

[Data binding directives 13](#_Toc153732386)

[Property binding using ngModel directive 13](#_Toc153732387)

[Event binding using ngModel directive 13](#_Toc153732388)

[Two-way binding using ngModel directive 13](#_Toc153732389)

[Class binding using ngClass directive 13](#_Toc153732390)

[Style binding using ngStyle directive 13](#_Toc153732391)

[Element binding directives 14](#_Toc153732392)

[Element binding using ngIf directive 14](#_Toc153732393)

[Element binding using ngSwitch directives 15](#_Toc153732394)

[Element binding using ngFor directive 16](#_Toc153732395)

[Writing a Directive 17](#_Toc153732396)

[Metadata 17](#_Toc153732397)

[Identifier 17](#_Toc153732398)

[Writing a Component 18](#_Toc153732399)

[Metadata 18](#_Toc153732400)

[Template 18](#_Toc153732401)

[Input and output properties 18](#_Toc153732402)

[Writing an event 19](#_Toc153732403)

[Writing a Pipe 20](#_Toc153732404)

[Writing a Service 20](#_Toc153732405)

[Writing a Module 21](#_Toc153732406)

[Metadata 21](#_Toc153732407)

[Declarations 21](#_Toc153732408)

[Root Module 22](#_Toc153732409)

[Dependency Injection 23](#_Toc153732410)

[Dependency creation 23](#_Toc153732411)

[Dependency injection 23](#_Toc153732412)

[Dependency scope 24](#_Toc153732413)

[Dependency injector 24](#_Toc153732414)

[Routing 25](#_Toc153732415)

[Forms 25](#_Toc153732416)

[HTTP (RxJS) 25](#_Toc153732417)

[Project Set up 26](#_Toc153732418)

[QuickStart seed 26](#_Toc153732419)

[Angular CLI 26](#_Toc153732420)

[Building Angular app 27](#_Toc153732421)

[TypeScript compilation 27](#_Toc153732422)

[Ahead-of-time (AOT) compilation 27](#_Toc153732423)

[Angular Universal 28](#_Toc153732424)

[More 28](#_Toc153732425)

[SystemJS 28](#_Toc153732426)

[webpack 28](#_Toc153732427)

# Building Blocks

## Directive

One of the defining features of a single page application is its manipulation of the DOM tree. Instead of serving a whole new page every time a user navigates, whole sections of the DOM appear and disappear according to the application state.

Angular has a powerful template engine that lets us easily manipulate the DOM structure of our elements.

A directive is an instruction to transform the DOM.

There are three kinds of Angular directives:

**Attribute** directive changes the appearance or behaviour of an existing DOM element.

ngModel, ngStyle, ngClass

**Structural** directive changes the DOM layout by adding and removing DOM elements.

ngIf, ngSwitch, ngFor

**Component** directive defines a custom HTML element.

A directive is a class adorned with the **@Directive** decorator function to attach metadata to the class.

## Template, Component, Data Binding

**Template**

A template is HTML + Angular syntax (i.e., data bindings, directives, custom HTML elements, pipes) that tells how to render a patch of screen called a *view*.

**Component**

A component's job is to enable the user experience. It mediates between the view (rendered by the template) and the application logic (which often includes some notion of a model). It presents properties and methods for data binding. It delegates everything nontrivial to services (and is thus easier to unit test with a mock service).

A component is a directive that defines a custom HTML element.

A component is a class adorned with the **@Component** decorator (that extends **@Directive** decorator function) to attach metadata to the class.

This class is associated with a template.

**Data Binding**

Data Binding is a mechanism to automatically push data values into the HTML controls (from component to element) and turning user responses into value updates and actions (from element to component).

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.

## Service

Service is a *broad* category that deals with application logic/data and cross-cutting concerns though each service has a *narrow*, well-defined purpose and meets it well.

It does not deal with presentation logic. It is not specific to any view. It is reusable – it is used across components. It is also used across other services.

Example – user input validation, tax calculator, data service, backend service, logging service, message bus, application configuration, etc.

Angular has no definition of a service. There is no service base class, and no place to register a service. A service is just a plain class (that can be adorned with the **@Injectable** decorator).

MVC/MVVM

View

**T**emplate

Controller / ViewModel

**C**omponent

Model and Application Logic

**S**ervice

View uses data binding to read its state from Controller; passes user input and actions to Controller.

Component class should be lean. It should mediate between the view 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 nontrivial to services.

Controller holds the state of View; receives input and user actions from view; delegates to Application Logic and updates the state of View with returned Model.

Controller doesn't validate user input, fetch data from the server, or log directly to the console. It delegates such tasks to services.

Service class should have a narrow, well-defined purpose. It should do something specific and do it well.

@ Business service, utility service.

A service may depend on other services.

Directive = JSP directive

Template = JSP file

Component = Java Bean

Service = Service

## Metadata, Dependency Injection

**Metadata**

Any plain class can be turned into an Angular Directive, Component, Pipe, dependent by attaching metadata to it.

Meta data can be attached to class fields as well.

In TypeScript, metadata is attached to a class using a **decorator**.

**Dependency Injection**

Dependency injection is a way to supply a new instance of a class with the fully-formed dependencies (i.e., instances of other classes) it requires. It lets you define many simple classes that each does one thing well but relies on other classes and then wire them together at runtime.

Most dependencies are services. Most dependents are components and services.

## Angular Module

Angular Modules help organize an Angular application into cohesive blocks of code, each dedicated to an application domain, a workflow, or a closely related set of capabilities.

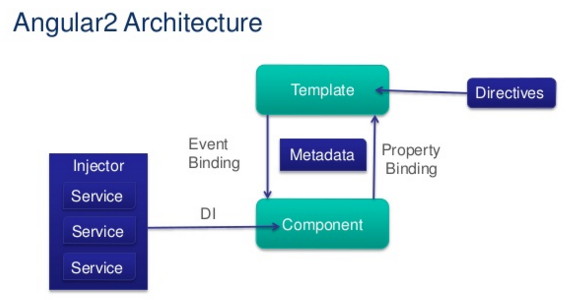
Every application has at least one Angular module, the root module (@ or rather bootstrap module), that is used to bootstrap the application.

While the root module may be the only module in a small application, most apps have many more feature modules.

An Angular Module groups together *view* classes (directives, components, and pipes). It may also include providers.

An Angular Module is a class adorned with the **@NgModule** decorator function to attach metadata to the class.

Architecture overview – <https://angular.io/docs/ts/latest/guide/architecture.html>



# Template Syntax

## HTML

All HTML syntax is valid template syntax, except, <html>, <body>, <base> and <script>.

## Template expression and statement

### Template Expression

A template expression produces a value that is assigned to a property of a binding target such as an element, directive or component.

Many JavaScript expressions are legal template expressions, but not all.

Following JavaScript syntax is not allowed:

assignments (=, +=, -=, ...)

increment and decrement operators (++ and --)

bitwise operators | and &

new

chaining expressions with ; or ,

Other notable differences from JavaScript syntax include:

template expression operators, such as ?. and |.

Template expressions cannot refer to anything in the global namespace. They can’t refer to window or document. They can’t call console.log or Math.max.

Template expressions are restricted to referencing members of the **expression context**, which is typically the component instance. The expression context can include objects other than the component. A template reference variable is one such alternative context object.

The following guidelines should be adhered to:

No side effects

A template expression should not change any application state (@ view should only read application state). Reading a display value must not change some other display value.

Idempotence

A template expression should always return exactly the same thing when called twice in a row during a single turn of the event loop.

Quick execution

A template expression should finish quickly. Consider caching values computed from other values when the computation is expensive.

Angular executes template expressions more often than we think. They can be called after every keypress or mouse move. If expressions don’t finish quickly, the user experience may drag, especially on slower devices.

Simplicity

A template expression should be simple – a property name or method call should be the norm. An occasional Boolean negation (!) is OK.

### Template Statement

A template statement responds to an event raised by a binding target such as an element, directive or component.

Following JavaScript syntax is not allowed:

operator assignments (+=, -=)

increment and decrement operators (++ and --)

bitwise operators | and &

new

However, the following is allowed:

basic assignment (=)

chaining expressions with ; or ,

But template expression operators, such as ?. and |, are not allowed.

Template statements cannot refer to anything in the global namespace. They can’t refer to window or document. They can’t call console.log or Math.max.

Template statements are restricted to referencing members of the **statement context**, which is typically the component instance. The statement context can include objects other than the component. A template reference variable is one such alternative context object. Also, the event object named $event representing the "message" or "payload" of the raised event can also be used.

The following guidelines should be adhered to:

Side effects

A template statement should update application state from user input. We're free to change anything, anywhere, during this turn of the event loop.

Simplicity

A template statement should be simple – a simple property assignment or method call should be the norm.

### Safe navigation operator

***object*?.*propertyObject*?.*property***

The above expression returns null when it hits the first null value, avoiding null reference error.

The above operator is useful in cases when null is a legitimate value (especially when we know the data will arrive eventually) and a blank should be displayed.

### Pipe

***value* | *pipe-1*:*param-a*:*param-b* | *pipe-2*:*param-a*:*param-b***

A pipe is a function that accepts an input value and returns a transformed value and is called with an operator-like syntax.

A pipe is a class adorned with the **@Pipe** decorator function to attach metadata to the class.

E.g., **json** pipe is particularly helpful for debugging our bindings.

### Template Reference Variable

<element **#*var*** >

<element **ref-*var*** >

<element **#*var*=”*directive*”** >

<element **ref-*var*=”*directive*”** >

A template reference variable is a reference to a DOM element or directive. It can be referred to anywhere in the current template.

## Data Binding

### Data Binding Overview

|  |  |  |  |
| --- | --- | --- | --- |
| **Data direction** | **Binding type** | **Binding Target\*** | **Binding Source** |
| One-way  from component to element | Interpolation | Element property  Directive property | String interpolated with template expression |
|  | Property | Element property  Component property  Directive property | Template expression |
|  | Attribute | Element attribute | Template expression |
|  | Class | Element’s class property | Template expression |
|  | Style | Element’s style property | Template expression |
| One-way  from element to component | Event | Element event  Component event  Directive event | Template statement |
| Two-way  from component to element and from element to component | Two-way (Property and Event) | Component property and event | Template expression |

Irrespective of data direction, element is called the binding target and appears on the left side of = and component is called the binding source and appears on the right of =.

\*Properties are defined by the DOM.

Some properties can be initialised and also set later.

Some properties can be initialised but not set later (e.g., table span, ARIA, SVG).

Some properties can’t be initialised but only set later (e.g., textContent).

Attributes are defined by HTML and are used to initialise DOM properties and then they are done.

<input id="name" type="text" value="Bob">

User enters "Sally" into the input box.

document.getElementById("name").getAttribute('value') //returns "Bob"

document.getElementById("name").value // returns "Sally"

~~Some HTML attributes initialize DOM properties and then they are done. Property values can change; attribute values can't.~~

~~<input id="name" type="text" value="Bob">~~

~~User enters "Sally" into the input box.~~

~~document.getElementById("name").getAttribute('value') //returns "Bob"~~

~~document.getElementById("name").value // returns "Sally"~~

~~Some DOM properties don't have corresponding HTML attributes.~~

~~Property data binding deals with properties, not attributes.~~

~~Some HTML attributes don't have corresponding DOM properties.~~

~~Attribute data binding deals with such attributes.~~

~~Why can’t Angular make out itself whether a HTML attribute corresponds to a DOM property or not and whether a DOM property corresponds to a HTML attribute or not so that we can use same syntax for property and attribute binding?~~

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

<p>Hello **{{*expression*}}**</p>

<div title="Hello **{{*expression*}}**">

Interpolation is used to bind an **element / directive property** to a calculated string between element tags or a calculated string within attribute assignments**.**

Interpolation is an alternative syntax for property binding. I won’t prefer it as it looks more like dynamic property initialization rather than property binding.

Also, property binding sanitizes the values (removes <script> tags from string values) before displaying them, while interpolation does not.

### Property binding

<input **[*property*]="*expression*"**>

<input **bind-*property*="*expression*"**>

Property binding is used to bind an element / component / directive property to a template expression.

The most common property binding binds an element property to a component property. Element properties may be the more common targets, but Angular looks first to see if the property name specified is a property of a known directive, e.g., ngModel, ngClass, ngStyle are properties of known directives.

<input [ngModel]="expression">

<div [ngClass]="{className1: expression1, className2: expression2}">

<div [ngStyle]="{style-prop1: expression1, style-prop2: expression2}">

### Attribute binding

<div **[attr.*attribute*]="*expression***">

Attribute binding is used to dynamically initialise an element property with a template expression that resolves to a string.

It creates an attribute and dynamically sets its value.

*Can we dynamically initialise a non-string property of a custom component?*

### Class binding

<div class="c1 c2 c3" **[class]="*expression*"**>

Class binding is used to toggle the presence of all classes on an element as per the truthiness of a template expression.

<div **[class.*className*]="*expression*"**>

Class binding is used to toggle the presence of a single class on an element as per the truthiness of a template expression.

### Style binding

<div **[style.*style-property*]="*expression*"**>

<div **[style.*styleProperty*]="*expression*"**>

Style binding is used to bind a single inline style on an element to a template expression.

### Event binding

<button **(*event*)="*statement*"**>

<button **on-*event*="*statement*"**>

Event binding is used to bind an element / component / directive event to a template statement.

The most common event binding binds an element event to a component method. Element events may be the more common targets, but Angular looks first to see if the event name specified matches an event property of a known directive, e.g., ngModelChange is an event property of a known directive.

<input (ngModelChange)="statement">

### Two-way binding (Property and Event binding)

<my-cmp **[(*property*)]="*expression*"**>

<my-cmp **bindon-*property*="*expression*"**>

Two-way binding is used to bind a component property and the corresponding property change event to a template expression and a template statement, respectively.

It is really just syntactic sugar for a property binding and an event binding equivalent to the following:

<my-cmp **[*property*]="*expression*" (*property*Change)="*expression*=$event.target.value"**>

It works for components that follow x property and xChange property change event naming convention.

Sadly, no native HTML element follows this naming convention.

However, we can follow this naming convention for a custom component that we write.

## Data binding directives

### Property binding using ngModel directive

<input **[ngModel]="*expression*"**>

<input **bind-ngModel="*expression*"**>

It binds element’s data property to a template expression.

It frees us from remembering the name of element’s data property.

### Event binding using ngModel directive

<input **(ngModelChange)="*statement*"**>

<input **on-ngModelChange="*statement*"**>

It binds element’s data property change event to a template statement.

It frees us from remembering the name of element’s data property change event.

### Two-way binding using ngModel directive

<input **[(ngModel)]="*expression*"**>

<input **bindon-ngModel="*expression*"**>

It binds element’s data property and data property change event to a template expression and a template statement, respectively.

It is equivalent to the following:

<my-cmp **[ngModel]="*expression*" (ngModelChange)="*expression*=$event.target.value"**>

ngModel directive works for component that are supported by a value accessor.

Specific form elements, such as the input text box, are supported by a ControlValueAccessor. So we need not remember names of those elements’ data property and data property change event and we can use two-way binding for them too.

For a custom component that we write, we can follow x / xChange naming convention making it easy to remember names of its data property and data property change event and enabling the use of basic two-way binding syntax. Thus it is unnecessary to write a value accessor for it.

For a custom component that we don’t control and that does not follow x / xChange naming convention, we may want to write a value accessor so that it frees us from remembering the name of component’s data property and data property change event as well as allows the use of two-way binding.

### Class binding using ngClass directive

<div **[ngClass]="{*className1*: *expression1*, *className2*: *expression2*}"**>

It toggles the presence of multiple classes on an element as per the truthiness specified by a map object where each key is a CSS class name and value is a boolean template expression.

### Style binding using ngStyle directive

<div **[ngStyle]="{*style-prop1*: *expression1*, *style-prop2*: *expression2*}"**>

It binds multiple inline styles on an element as per a map object where each key is a CSS style name and value is a template expression.

## Element binding directives

### Element binding using ngIf directive

<div **\*ngIf="*expression*"**>...</div>

<div **template=”ngIf:*expression*"**>...</div>

<template **[ngif]=”*expression*”**>

<div>...<div>

</template>

It binds presence of an element in DOM to a template expression.

It adds to or removes from DOM an element subtree based on whether the template expression is a truthy value (not null / true) or a falsy value (null / false).

@ Conditional logic applied to view structure (as opposed to program flow) and bound to controller.

When the template expression is null / false, Angular physically removes the element subtree from the DOM. It destroys elements in the subtree, along with their state. It stops to check for changes that could affect data bindings. The subtree can be garbage-collected and resources are freed up.

When the template expression is not null / true, Angular recreates the element subtree and elements have to re-initialize – this is expensive as data that had been in memory just moments ago has to be re-fetched, unless it has been cached in a companion service.

We may show or hide an element tree with a class or style binding.

When an element subtree is hidden, it remains in the DOM. Elements in the subtree are preserved, along with their state and behaviour. It continues to listen to events. Angular continues to check for changes that could affect data bindings. The subtree continues to tie up resources.

When an element subtree is shown, the elements don’t have to re-initialize. Showing the element again is very quick.

The show/hide technique is probably fine for small element trees and when we have to toggle the element tree very frequently. In general it is best to use ngIf.

### Element binding using ngSwitch directives

<container-element **[ngSwitch]="*expression*"**>

<element **\*ngSwitchCase="*case1Expression*"**>...</element >

<element **\*ngSwitchCase="*case2Expression***">...</element>

<element **\*ngSwitchDefault>**...</element>

</container-element>

<container-element **[ngSwitch]="*expression*"**>

<element **template=”ngSwitchCase:*case1Expression*"**>...</element >

<element **template=”ngSwitchCase:*case2Expression***">...</element>

<element **template=”ngSwitchDefault”>**...</element>

</container-element>

<container-element **[ngSwitch]="*expression*"**>

<template **[ngSwitchCase]=”*case1Expression*”**>

<element>...</element >

</template>

<template **[ngSwitchCase]=”*case2Expression*”**>

<element>...</element >

</template>

<template **ngSwitchDefault**>

<element>...</element >

</template>

</container-element>

It binds selection of an element from a set of possible elements to a template expression.

At any particular moment, at most one of these elements is in the DOM.

### Element binding using ngFor directive

<li **\*ngFor="let *item* of *itemList*"**>

It binds repetition of an element to a list in the expression context.

The let keyword creates a template input variable *item* (which is not the same as a template reference variable) and can be used within the element / template (?).

<li **\*ngFor="let *item* of *itemList*; let *i*=index"**>

Here, another template input variable *i* is created that captures the index.

<li **\*ngFor="let *item* of *itemList*; let *f*=first; let *l*=last; let *o*=odd; let *e*=even;"**>

Here, more template input variables *f*, *l,* *o* and *e* are created.

<li **\*ngFor="let *item* of *itemList*; trackBy:*trackByFunction*"**>

When the item list changes, Angular makes the following changes to DOM:

When an item is added, a new instance of the element is added to the DOM.

When an item is removed, its element instance is removed from the DOM.

When items are reordered, their respective elements are reordered in the DOM.

@ When a property of an existing item has changed, the corresponding existing DOM element is updated.

Angular uses object identity to identify an item. It is possible for the identities of items to change while the data does not. This can happen, for example, if the list is produced from an RPC to the server, and that RPC is re-run. Even if the data hasn't changed, the second response will produce items with different object identities, and Angular will tear down all the elements in DOM and rebuild it (as if all old elements were deleted and all new elements inserted). This is an expensive operation and should be avoided if possible.

The ngFor directive can be provided a tracking function which has two arguments: index and *item*. Angular then identifies an item with the return value of the tracking function.

<li **\*ngFor="let *item* of *items*; let *i*=index; trackBy:*trackByFn*"**>...</li>

<li **template="ngFor let *item* of *items*; let *i*=index; trackBy:*trackByFn*"**>...</li>

<template **ngFor let-*item* [ngForOf]="*items*" let-*i*="index" [ngForTrackBy]="*trackByFn*"**>

<li>...</li>

</template>

Note that instead of semi-colon separator above, comma or space separator could also be used.

Template Syntax – <https://angular.io/docs/ts/latest/guide/template-syntax.html>

# Writing a Directive

## Metadata

A directive is a class adorned with the **@Directive** class decorator function that takes a single metadata object whose properties describe the component. The most important properties are discussed below.

### Identifier

@Directive({

**selector: '*myDirective*'**

})

class *MyDirective* {

}

**selector** specifies a CSS selector that identifies the directive within a template. Supported selectors include element, [attribute], .class, and :not().

Does not support parent-child relationship selectors.

TBD:

Writing a directive –

<https://angular.io/docs/ts/latest/guide/attribute-directives.html#write-directive>

<https://angular.io/docs/ts/latest/guide/structural-directives.html#unless>

# Writing a Component

## Metadata

A component is a class adorned with the **@Component** class decorator function (that extends **@Directive** class decorator function) that takes a single metadata object whose properties describe the component. The most important properties are discussed below.

### Template

@Component({

**template: '*<div>...</div>*'**

})

class *MyComponent* {

}

@Component({

**templateUrl: '*my.component.html*'**

})

class *MyComponent* {

}

### Input and output properties

@Component({

**inputs: ['*inProp1*:*ip1Alias*', '*inProp2*:*ip2Alias*'],**

**outputs: ['*outProp1:op1Alias*', '*outProp2:op2Alias*']**

})

class *MyComponent* {

}

@Component({

})

class *MyComponent* {

**@Input(‘*ip1Aias*’) *inProp1;***

**@Input(‘*ip2Aias*’) *inProp2;***

**@Output(‘*op1Alias*’) *outProp1;***

**@Output(‘*op2Alias*’) *outProp2;***

}

Note: Either use the class decorator’s inputs and outputs arrays or the class field decorators but not both.

A template and the accompanying component together define a custom HTML element. The template itself may use another custom HTML element.

In a template, the data binding source (that appear on the right side of =) are properties of the *accompanying component* and are automatically available for binding while the data binding target (that appear on the left side of =) are properties of the *child component* and have to be explicitly marked as target.

Target properties are either inputs or outputs of the component. Input properties are targets of property binding and receive data values. Output properties are targets of an event binding and expose event producers, such as EventEmitter objects.

Sometimes we want the public name of an input/output property to be different from the internal name. This is frequently the case with attribute directives where the property name should be the same as the selector but often selector makes for a poor property name.

The public name can be specified by declaring an alias for the property.

@ For a custom HTML element, the properties of type string can be initialised by the corresponding attributes using the same syntax as that for standard HTML elements.

## Writing an event

@Component({

template: `

<div>

<button (click)="delete()">Delete</button>

</div>`

})

class HeroDetailComponent {

hero: Hero;

**@Output() deleteRequest = new EventEmitter<Hero>();**

**delete() {**

**this.deleteRequest.emit(this.hero);**

**}**

}

Components typically raise custom events with an Angular EventEmitter. The component creates an EventEmitter and exposes it as an output target property and calls EventEmitter.emit(*payload*) in an event handler of its own.

How to decide whether the component should itself raise an event or not while handling an event that it has subscribed to?

<https://angular.io/docs/ts/latest/guide/template-syntax.html#inputs-outputs>

<https://angular.io/docs/ts/latest/guide/template-syntax.html#event-binding> > Custom events with EventEmitter

[https://angular.io/docs/ts/latest/guide/glossary.html#!#component](https://angular.io/docs/ts/latest/guide/glossary.html#component)

<https://angular.io/docs/ts/latest/guide/cheatsheet.html>

<https://angular.io/docs/ts/latest/api/core/index/Component-decorator.html>

TBD:

<https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>

<https://angular.io/docs/ts/latest/guide/component-styles.html>

<https://angular.io/docs/ts/latest/cookbook/component-relative-paths.html>

<https://angular.io/docs/ts/latest/cookbook/component-communication.html>

# Writing a Pipe

TBD:

<https://angular.io/docs/ts/latest/guide/pipes.html>

<https://angular.io/guide/pipes#example-transforming-a-value-exponentially>

<https://www.npmjs.com/package/ng2-filter-pipe>

# Writing a Service

@ Write to an interface.

If we were actually getting data from a remote server, the API would have to be asynchronous, perhaps returning a Promise.

Providers can be registered with injector in components (recall that components are dependents) or modules but not in services (recall that services can be dependents as well).

<https://angular.io/docs/ts/latest/guide/architecture.html#services>

<https://angular.io/docs/ts/latest/tutorial/toh-pt4.html>

<https://angular.io/docs/ts/latest/guide/glossary.html#S>

https://angular.io/docs/ts/latest/guide/dependency-injection.html#angular-dependency-injection

# Writing a Module

## Metadata

An Angular Module is a class (@ an empty class) adorned with the **@NgModule** class decorator function that takes a single metadata object whose properties describe the module. The most important properties are discussed below.

### Declarations

@NgModule({

**declarations: [ *MyDirective*, *MyComponent*, *MyPipe* ]**

})

class *MyModule* {

}

imports - other Angular modules whose exported classes are needed by component **templates** declared in this Angular module.

declarations - the **view classes** that belong to this Angular module. Angular has three kinds of view classes: components, directives, and pipes.

exports - the subset of declared classes, the subset of imported modules as well as any other module that should be visible and usable in the component **templates** of other modules.

providers - creators of services that this module contributes to the global collection of services; they become accessible in all parts of the app.

bootstrap - the main application view, called the root component, that hosts all other app views. Only the root module should set this bootstrap property.

To use a **class** in your **class**, you need to do the following:

Include the corresponding package in package.json and install it. May be make an entry in the map section of ‘systemjs.config.js’ that contains the “systemjs” configurations.

Use the TS import statement in your class to import that class.

To use a **directive**/**component/pipe** in your **template** anywhere in your module, you need to do the following:

Include the corresponding package in package.json and install it. May be make an entry in the map section of ‘systemjs.config.js’ that contains the “systemjs” configurations.

Use the TS import statement in your class to import the corresponding module/class in your module class.

Use the Angular module import statement to import the corresponding module/class in your module class.

@angular/common – ngIf, ngSwitch

@angular/core – Directive, Component, NgModule

## Root Module

The root module can be called anything you want – the conventional name is AppModule.

<https://angular.io/docs/ts/latest/guide/architecture.html#modules>

<https://angular.io/docs/ts/latest/guide/ngmodule.html>

[https://angular.io/docs/ts/latest/guide/glossary.html#!#M](https://angular.io/docs/ts/latest/guide/glossary.html#M)

<https://angular.io/docs/ts/latest/guide/glossary.html#angular-module>

<https://angular.io/docs/ts/latest/guide/cheatsheet.html>

<https://angular.io/docs/ts/latest/api/core/index/NgModule-interface.html>

<https://angular.io/docs/ts/latest/guide/appmodule.html>

# Dependency Injection

Angular ships with its own dependency injection framework. This framework can also be used as a standalone module by other applications and frameworks.

## Dependency creation

A provider provides a recipe for creating a new instance of a dependency.

**providers: [{provide: *token*, useClass: *superClass*}]**

Here *token* is dependency class and we provide a superclass of dependency class as a *recipe*.

**providers: [*token*]**

Here *token* is dependency class and we provide the dependency class itself as a *recipe*.

**providers:[{provide: *token*, useFactory: *factory*}]**

Here *token* is dependency class and we provide a factory as a recipe.

**providers: [{provide: *token*, useValue: *object*}]**

Here *token* is dependency class and we provide a ready-made object rather than ask the injector to create it from a class.

**providers:[{provide: *token*, useClass: *recipe*}, {provide: *aliasToken*, useExisting: *token*}]**

Here *token* is dependency class and *aliasToken* is another dependency class and we provide the same *recipe* for *aliasToken* as that for *token*.

Non-class dependencies

Sometimes the thing we want to inject is a string, function, or object.

<https://angular.io/docs/ts/latest/guide/dependency-injection.html#dependency-injection-tokens>

## Dependency injection

A dependent is a class adorned with the **@Injectable() marker** decorator function.

Note that @Directive(), @Component() and @Pipe() are subtypes of **@Injectable()**.

It is recommend to add **@Injectable()** to every service class, even those that don't have dependencies and, therefore, do not technically require it. Here's why:

Future proofing: No need to remember **@Injectable()** when we add a dependency later.

Consistency: All services follow the same rules, and we don't have to wonder why a decorator is missing.

Injection is through constructor.

Injection is done using autowiring by type.

Optional dependencies

constructor(**@Optional()** private logger: Logger) {

if (this.logger) {

this.logger.log(some\_message);

}

}

## Dependency scope

Providers can be registered with injector in components (recall that components are dependents) or modules but not in services (recall that services can be dependents as well).

Registering in a component means you get a new instance of the dependency with each new instance of that component and that same instance of the dependency is available to component’s children.

Registering in ~~the root~~ module means the same instance of the dependency is available everywhere.

Should I add other providers to a module or a component?

Should I add app-wide providers to the root AppModule or the root AppComponent?

<https://angular.io/docs/ts/latest/cookbook/ngmodule-faq.html#q-root-component-or-module>

## Dependency injector

*Providers* are registered with *injectors*.

An *injector* can create a new dependency instance from a *provider*.

Injectors are also responsible for instantiating dependents.

Angular creates and delivers dependencies "just-in-time".

When Angular creates a component, it first asks an ***injector*** for the dependencies that the component requires. An *injector* maintains a container of dependency instances that it creates. If a requested dependency instance is not in the container, the *injector* makes one and adds it to the container before returning the dependency to Angular. When all requested dependencies have been returned, Angular can call the component's constructor with those dependencies as arguments. This is dependency injection.

Working with injectors directly

<https://angular.io/docs/ts/latest/guide/dependency-injection.html>

Injector hierarchy

<https://angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html>

Note: Having one class per file is a recommended practice. However, if we still choose to define a component and its service dependency in the same file, defined the service before the component; otherwise we'll get a runtime null reference error.

<https://angular.io/docs/ts/latest/guide/architecture.html#dependency-injection>

<https://angular.io/docs/ts/latest/guide/dependency-injection.html>

<https://angular.io/docs/ts/latest/guide/hierarchical-dependency-injection.html>

<https://angular.io/docs/ts/latest/cookbook/dependency-injection.html>

<https://angular.io/docs/ts/latest/cookbook/ngmodule-faq.html#q-component-or-module>

<https://angular.io/docs/ts/latest/cookbook/ngmodule-faq.html#q-root-component-or-module>

<https://angular.io/docs/ts/latest/guide/glossary.html#D>

https://angular.io/docs/ts/latest/guide/cheatsheet.html

<https://angular.io/docs/ts/latest/api/core/index/Injectable-decorator.html>

# Routing

<base href="/"> -- href specifies the path root for the router. “/” means “app/”.

Configure path-to-component mapping in module file.

In the host template (i.e., module’s top most component’s template), put the RouterOutlet tag at the place where the navigated view should be displayed:

<router-outlet></router-outlet>

<!-- Routed views go here -->

In the templates, define navigation links (anchor elements) using routerLink attributes.

# Forms

Template reference variable directive ngForm:

<form (ngSubmit)="onSubmit(***theForm***)" **#*theForm*="ngForm"**>

<button type="submit" [disabled]="!**theForm**.form.valid">Submit</button>

</form>

It lets you create a template reference variable that is not a reference to the native form element but to an object that wraps the native form element and endows it with additional superpowers such as the ability to track the validity of user input.

# HTTP (RxJS)

Publishers:

Reactor Mono/Flux

RxJava Single/Observable

RxJS Observable – zero or more events

JavaScript native Promise – single event

RxJS Observable v/s JavaScript native Promise:

With Observable it doesn't matter if you want to handle 0, 1, or multiple events. You can utilize the same API in each case.

Observable provides the features of Promise and more – retry, cancellation, etc.

While a Promise starts immediately, an Observable only starts if you subscribe to it. This is why Observables are called lazy.

<https://stackoverflow.com/a/37365955/6350321>

Observable.map()

Observable.catch()

Observable.subscribe()

<https://angular.io/docs/ts/latest/guide/server-communication.html>

<https://angular.io/docs/ts/latest/tutorial/toh-pt6.html>

# Angular Material

# Bootstrap

How do you embed bootstrap? -- install and then enter in angular.json's styles and scripts section; <https://blog.hubspot.com/website/add-bootstrap-to-angular-2>

# Testing

app.component.ts

app.component.spec.ts

## Jasmine

<https://jasmine.github.io/tutorials/your_first_suite>

Use appropriate string parameters with ‘describe’ and ‘it’ for your specs to read as full sentences in traditional BDD style.

describe("A calculator", function() {  
 var calculator;  
  
 beforeEach(function() {  
 calculator = new Calculator();  
 });  
  
 it("should be able to add two numbers", function() {  
 expect(calculator.add(2, 2)).toBe(4);  
 });  
  
 it("should be able to subtract two numbers", function() {  
 expect(calculator.subtract(2, 2)).toBe(0);  
 });

});

While the authors of Jasmine have intended it as a BDD testing framework, it can also be used with TDD and unit testing.

<https://www.testim.io/blog/is-jasmine-bdd-or-tdd/>

## Mocha

Mocha does not include an assertion library, which gives the developer the choice to choose one according to their preference. Mocha also provides more flexibility in configuration and integration with other tools, making it a better choice for projects with complex testing requirements.

<https://mochajs.org/#getting-started>

## QUnit

## Karma

You can run test cases with Node or with JavaScript engine of different browsers.

Karma lets you run the test cases in a browser.

Karma has advantages to other test runners to improve your TDD/BDD development cycle. It "watches" files, so when you save a change, Karma runs tests and reports back instantly, no switching context to Web Browser to run the test.

<https://stackoverflow.com/a/36427079/6350321>

<https://stackoverflow.com/a/34249126/6350321>

<https://www.youtube.com/watch?v=bJc078szrZA>

# Project Set up

## QuickStart seed

Pre-requisites:

Install node v4.x.x or higher and npm 3.x.x or higher.

Create project:

Download the QuickStart seed from <https://github.com/angular/quickstart> and unzip it into your project folder.

Non-essential files listed in ‘non-essential-files.txt’ may be deleted.

Install npm packages:

Install npm packages – npm install

Launch the app:

npm start

TODO:

Understand project setup anatomy:

Source code files

Tool files

Tool usage

<https://angular.io/docs/ts/latest/guide/setup.html>

<https://angular.io/docs/ts/latest/guide/setup-systemjs-anatomy.html>

<https://angular.io/docs/ts/latest/guide/deployment.html>

<https://github.com/angular/quickstart/blob/master/README.md>

## Angular CLI

Pre-requisites:

Install node v6.9.x or higher and npm 3.x.x or higher.

Install Angular CLI globally – npm install -g @angular/cli

Create project and install npm packages:

ng new my-app

Launch the app:

cd my-app

ng serve --open

<https://angular.io/docs/ts/latest/cli-quickstart.html>

<https://cli.angular.io/>

<https://github.com/angular/angular-cli/wiki>

# Building Angular app

## TypeScript compilation

## Ahead-of-time (AOT) compilation

@Before the browser can render an Angular application, the Angular syntax must be converted to executable JavaScript by the Angular compiler.

Ahead-of-time (AOT) vs just-in-time (JIT):

There is actually only one Angular compiler. The difference between AOT and JIT is a matter of timing and tooling.

Ahead-of-time (AOT) compilation -- Angular compiler runs once at build time using one set of libraries.

Just-in-time (JIT) compilation -- Angular compiler runs every time for every user at runtime using a different set of libraries.

Shortcomings of JIT:

The application is bigger because it includes the Angular compiler and a lot of library code that the application won't actually need. Bigger apps take longer to transmit and are slower to load.

Views take longer to render because of the in-browser compilation step.

Angular syntax errors are discovered at runtime, which is late in the process.

Advantages of AOT:

AOT compilation converts more of the application to JavaScript, which in turn makes more of the application "tree shakable".

The compiler discards unused Angular directives that a tree-shaking tool can then exclude.

The compiler inlines external HTML templates and CSS style sheets within the application JavaScript, eliminating separate ajax requests for those source files.

AOT compiles HTML templates and components into JavaScript files long before they are served to the client. With no templates to read and no risky client-side HTML or JavaScript evaluation, there are fewer opportunities for injection attacks.

AOT radically improves performance by compiling ahead-of-time (AOT) during a build process.

If you want your app to be usable on slow networks, AOT is a must. Universal should be considered if first impressions are important.

In the JIT-compiled app, main.ts is the entry point that uses platformBrowserDynamic to bootstrap the app module.

In the AOT-compiled app, main-aot.ts is the entry point that uses platformBrowser to bootstrap an app module factory created by pre-compiling the app module.

npm install @angular/compiler-cli@4.0.0 @angular/platform-server@4.0.0 --save

<https://angular.io/docs/ts/latest/cookbook/aot-compiler.html>

## Angular Universal

Server-side Rendering for Angular 2 apps.

Rendering to an HTML string on the server instead of the browser DOM.

Lets you run Angular application on any Node.js server.

The resulting app is necessarily limited. For example, a Universal app does not handle browser events such as mouse or keyboard inputs, nor send AJAX requests. An available tool called Preboot will record browser events so they can be played back into the full Angular app once it is loaded.

Why do it? There are two main reasons:

SEO / No JavaScript

Startup performance

<https://angular.io/docs/ts/latest/guide/universal.html>

<https://universal.angular.io/>

<http://stackoverflow.com/questions/31079695/angular-js-benefits-over-jsp-view>

<http://blog.jhades.org/the-java-origins-of-angular-js-angular-vs-jsf-vs-gwt/>

## More

Pruned libraries

Tree shaking

Inlining: pulls template html and css into the components.

Bundling

Minification

Uglification

## SystemJS

## webpack

Install angular2-template-loader.

You'll need polyfills to run an Angular application in most browsers. Polyfills should be bundled separately from the application and vendor bundles. Add a polyfills.ts to the src/ folder.

There is an official webpack plugin for AOT (<https://github.com/angular/angular-cli/tree/master/packages/%40ngtools/webpack>). This plugin transpiles the TypeScript application code, bundles lazy loaded NgModules separately, and performs AOT compilation — without any changes to the source code.

TODO:

While keeping SystemJS, bring in webpack (create index-systemjs.html).

While keeping JIT, bring in AOT.

Bring in tree-shaking.

Angular Features

Develop Across All Platforms

Learn one way to build applications with Angular and reuse your code and abilities to build apps for any deployment target. For web, mobile web, native mobile and native desktop.

Convert Your Angular Application to a Native Mobile App (Android and iOS):

Using Angular CLI and Capacitor.js

<https://betterprogramming.pub/how-to-convert-your-angular-application-to-a-native-mobile-app-android-and-ios-c212b38976df>

<https://nativescript.org/>

<https://docs.nativescript.org/tutorials/build-a-master-detail-app-with-angular>

Ionic

Speed & Performance

Achieve the maximum speed possible on the Web Platform today, and take it further, via Web Workers and server-side rendering.

Angular puts you in control over scalability. Meet huge data requirements by building data models on RxJS, Immutable.js or another push-model.

Angular JS 2.0  
Starting with Angular  
• Introductions  
o Oasis Digital background  
o Class objectives and roadmap  
o Student experience and expectations  
• Preparation  
o Prerequisites review  
o What is Angular?  
o Angular motivations  
o Single page apps vs server-side web applications  
o Why Angular?  
o Where Angular fits  
o Our focus, Angular for complex, rich, enterprise application  
development  
• Getting started with Angular  
o ES5, ES2016, ESnext, Typescript, Angular  
o Simplest Angular application  
o What about CSS?  
o Typescript  
o Development tooling for class use  
• The Typescript module system  
o Dividing a program into modules / files  
• Angular templates and bindings  
o Template binding expressions  
o Null coalescing / safe traversal  
• Events and event bindings  
• Nested components  
o In-browser inspector tooling  
• Routing introduction  
o Generations of Angular routing  
o Routing to components  
o Configuring the router  
o HTML5 vs hash routes  
  
Creating your Angular project  
• Build tooling  
o SystemJS, optionally with JSPM  
o WebPack  
o Angular CLI  
o Build tooling at the start versus tooling in a mature enterprise  
project  
• Starting a project with Angular CLI  
• Component hierarchy and wireframing  
• Essential built-in directives  
o Branching with ngIf  
o Iteration with ngFor  
• Class and style bindings  
o [class.x] bindings  
o [style.y] bindings  
o [ngClass] for more complex class selection  
• Component styling  
o Scoped CSS  
o View Encapsulation  
o Targeting the component itself  
• Angular pipe introduction  
o Using the built-in pipes  
• Component data Binding  
o Bind data "downward" with @Input()  
o Use events to send data upward with @Output()  
• Dependency injection  
o Injection introduction  
o Why dependency injection?  
o What can be injected?  
o Explaining DI  
  
Asynchronous data and user interaction  
• Reactive forms  
o Standalone reactive controls  
o Control groups  
o Form Builder  
o ngSubmit  
• Observable form values - an introduction to Observables  
• Creating services  
o Injecting services into services  
• The async pipe  
o Asynchronous data - almost automatic  
o Capturing the async pipe data?  
• Loading data via HTTP  
o Show async data via the async pipe  
o Inject HTTP in to a service, not in to a component  
o Setting request headers  
o CORS discussion  
• Route parameters  
• Observable bindings  
o Observables in more depth  
o Route-driven observable data loading  
• The smart/display component pattern  
o Motivations  
o Implementation  
• Observable Composing and Chaining  
o Third-party API example  
o Observable error handling  
o Filtered list example  
  
Angular at scale - modules and lazy loading  
• Nested routing  
• Multiple NgModules  
o Inter-module dependencies  
o entryComponents  
• Lazy loading  
o Routes as the unit of lazy loading  
o Why?  
o How?  
o Preloading: eager lazy routes  
  
Data flow in an Angular application  
• The component lifecycle  
• Sharing data via binding  
o Bindings to adjacent component data  
o Bindings to data in a service  
• Change detection strategies - and why it matters  
o How change detection works  
o Change detection modes  
o Zone.JS  
• Centralized state management with hierarchical DI  
o State in observable in a service  
• NgRx / Redux Architecture  
o An industry-wide idea  
o History  
o Angular implementation  
  
Ancillary Angular features  
• Complex forms  
o Custom validators  
o Custom async validators  
• Custom pipes  
• Complex routing  
o Auxiliary routes  
o Route guards - protecting routes  
o Resolve - loading data during routing  
• Decorator directives  
o Introduction  
o Examples  
• NgContent (was transclusion)  
o Multiple Ng-Content  
• ngSwitch  
• Deeper with Observables  
o Wrap an API in Observables  
o Loading data with retry  
• Sanitization - secure handling of user-provided data  
• Unit testing  
o Unit testing at the command line (already discussed)  
o Unit testing in the browser  
• Multi-providers  
  
Upgrading, third-party, and legacy code  
• Using Angular in ES5  
• Upgrading from AngularJS 1.x  
o ngUpgrade  
o ngForward  
o Where did 1.x features go?  
• ES2015 code  
o History of JS  
• TypeScript examples  
o TypeScript decorators  
• More Observable error handling  
• Promises - still available, still useful  
• HTTP API access via promises  
• Two-way binding with ngModel  
o ngModel with more control types  
• Template forms  
• Tooling with Webpack - Instructor notes

<https://angular.io/docs/ts/latest/>

Docs Overview – <https://angular.io/docs/ts/latest/guide/>

Learning Path – <https://angular.io/docs/ts/latest/guide/learning-angular.html>

Quick Start – <https://angular.io/docs/ts/latest/quickstart.html>

Guide – <https://angular.io/docs/ts/latest/guide/>

Architecture overview – <https://angular.io/docs/ts/latest/guide/architecture.html>

Template Syntax – <https://angular.io/docs/ts/latest/guide/template-syntax.html>

Advanced – <https://angular.io/docs/ts/latest/guide/ngmodule.html>

Tutorial – <https://angular.io/docs/ts/latest/tutorial>

Cookbook – <https://angular.io/docs/ts/latest/cookbook/>

Style Guide – <https://angular.io/docs/ts/latest/guide/style-guide.html>

Glossary – <https://angular.io/docs/ts/latest/guide/glossary.html>

Cheat Sheet – <https://angular.io/docs/ts/latest/guide/cheatsheet.html>

API Reference – <https://angular.io/docs/ts/latest/api/>

<https://github.com/angular/angular.io/issues/2651>

<http://stackoverflow.com/questions/16263195/how-to-run-angularjs-documentation-localy>

<https://angular.io/resources/>

<https://www.ng-book.com/2/>

<https://www.manning.com/books/angular-2-in-action>

<https://www.manning.com/books/angular-2-development-with-typescript>

<https://leanpub.com/angular2-book>

<https://leanpub.com/practical-angular-2>

<https://angular-2-training-book.rangle.io/>

<http://www.angular2.com/>

TODO next:

TypeScript

Revision

Forms

Routing

HTTP and RXJS

Karma, Jasmine

NPM, WebPack

Angular Material

Native desktop (Electron), Native mobile (Ionic Framework, React Native, NativeScript)