

## 01. Components

Components are building blocks of an Angular application. A component is a combination of HTML template and a TypeScript (or a JavaScript) class. To create a component in Angular with TypeScript, create a class and decorate it with the Component decorator.

Consider the sample below:

```
import { Component } from '@angular/core';
@Component({
  selector: 'hello-ng-world',
  template: `<h1>Hello Angular world</h1>`
})
export class HelloWorld {
}
```

Component is imported from the core angular package. The component decorator allows specifying metadata for the given component. While there are many metadata fields that we can use, here are some important ones:

**selector:** is the name given to identify a component. In the sample we just saw, hello-ng-world is used to refer to the component in another template or HTML code.

**template:** is the markup for the given component. While the component is utilized, bindings with variables in the component class, styling and presentation logic are applied.

**templateUrl:** is the url to an external file containing a template for the view. For better code organization, template could be moved to a separate file and the path could be specified as a value for templateUrl.

**styles:** is used to specific styles for the given component. They are scoped to the component.

**styleUrls:** defines the CSS files containing the style for the component. We can specify one or more CSS files in an array. From these CSS files, classes and other styles could be applied in the template for the component.

## 02. Data-binding

String interpolation is an easy way to show data in an Angular application. Consider the following syntax in an Angular template to show the value of a variable.

```
`<h1>Hello {{title}} world</h1>`
```

**title** is the variable name. Notice single quotes (backtick `) around the string which is the ES6/ES2015 syntax for mixing variables in a string.

The complete code snippet for the component is as follows:

```
import { Component } from '@angular/core';
@Component({
```

```

    selector: 'hello-ng-world',
    template: `<h1>Hello {{title}} world</h1>`
  })
  export class HelloWorld {
    title = 'Angular 4';
  }

```

To show the value in a text field, use DOM property “*value*” wrapped in square brackets. As it uses a DOM attribute, it’s natural and easy to learn for anyone who is new to Angular. “*title*” is the name of the variable in the component.

```
<input type="text" [value]="title">
```

Such a binding could be applied on any HTML attribute. Consider the following example where the title is used as a placeholder to the text field.

```
<input type="text" [placeholder]="title" >
```

## 03. Events

To bind a DOM event with a function in a component, use the following syntax. Wrap the DOM event in circular parenthesis, which invokes a function in the component.

```
<button (click)="updateTime()">Update Time</button>
```

### 3.1 Accepting user input

As the user keys-in values in text fields or makes selections on a dropdown or a radio button, values need to be updated on component/class variables.

We may use **events** to achieve this.

The following snippet updates value in a text field, to a variable on the component.

```

<!-- Change event triggers function updateValue on the component -->
<input type="text" (change)="updateValue($event)">
  updateValue(event: Event){
    // event.target.value has the value on the text field.
    // It is set to the label.
    this.label = event.target.value;
  }

```

**Note:** Considering it’s a basic example, the *event* parameter to the function above is of type *any*. For better type checking, it’s advisable to declare it of type *KeyboardEvent*.

### 3.2 Accepting user input, a better way

In the Accepting user input sample, *\$event* is exposing a lot of information to the function/component. It encapsulates the complete DOM event triggered originally. However, all the function needs is the value of the text field.

Consider the following piece of code, which is a refined implementation of the same. Use template reference variable on the text field.

```
<input type="text" #label1 (change)="updateValue(label1.value)">
```

Notice *updateValue* function on *change*, which accepts value field on label1 (template reference variable). The update function can now set the value to a class variable.

```
updateValue(value: any){  
  // It is set to the label.  
  this.label = value;  
}
```

### 3.3 Banana in a box

From Angular 2 onwards, *two way data binding* is not implicit. Consider the following sample. As the user changes value in the text field, it's doesn't instantly update the title in h1.

```
<h1> {{title}} </h1>  
<input type="text" [value]="title">
```

Use the following syntax for two-way data binding. It combines value binding and event binding with the short form – `[( )]`. On a lighter note, it's called banana in a box.

```
<h1> {{title}} </h1>  
<input type="text" [(ngModel)]="title" name="title">
```

## 04. ngModel and form fields

ngModel is not only useful for two-way data binding, but also with certain additional CSS classes indicating state of the form field.

Consider the following form fields - first name and last name. ngModel is set to fname and lname fields on the view model (the variable vm defined in the component).

```
<input type="text" [(ngModel)]="vm.fname" name="firstName" #fname required />  
{{fname.className}} <br />  
<input type="text" [(ngModel)]="vm.lname" name="lastName" #lname /> {{lname.  
className}}
```

Please note, if ngModel is used within a form, it's required to add a name attribute. The control is registered with the form (parent using the name attribute value. Not providing a name attribute will result in the following error.

If ngModel is used within a form tag, either the name attribute must be set or the form control must be defined as 'standalone' in ngModelOptions.

It adds the following CSS classes as and when the user starts to use the form input elements.

**ng-untouched** – The CSS class will be set on the form field as the page loads. The user hasn't used the field and didn't set keyboard focus on it yet.

**ng-touched** – The CSS class will be added on the form field as user sets focus by moving keyboard cursor or clicks on it. Once the user moves away from the field, this class is set.

**ng-pristine** – This class is set as long as the value on the field hasn't changed.

**ng-dirty** – This class is set while user modifies the value.

**ng-valid** – This class is set when all form field validations are satisfied, none failing. For example a required field has a value.

**ng-invalid** – This class is set when the form field validations are failing. For example, a required field doesn't have a value.

This feature allows customizing CSS class based on a scenario. For example, an invalid input field could have red background highlighting it for the user.

Define the following CSS for the given component with form elements.

```
.ng-invalid{  
  background: orange;  
}
```

Check out figure 1. It depicts two form fields with CSS classes, indicating state of the form field next to it.

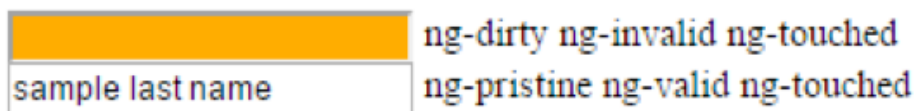


Figure 1: Input fields and CSS classes applied on it

## 05. ngModule

Create an Angular module by using the **ngModule** decorator function on a class.

A module helps package Angular artifacts like components, directives, pipes etc. It helps with ahead-of-time (AoT) compilation. A module exposes the components, directives and pipes to other Angular modules. It also allows specifying dependency on other Angular modules.

Consider the following code sample.

**Note:** All components have to be declared with an Angular module.

```
import { NgModule }      from '@angular/core';  
import { BrowserModule } from '@angular/platform-browser';  
import { MyAppComponent } from './app.component';  
  
@NgModule({  
  imports:      [ BrowserModule ],  
  declarations: [ MyAppComponent ],  
  bootstrap:    [ MyAppComponent ]  
})  
export class AppModule { }
```

Import **ngModule** from Angular core package. It is a decorator function that can be applied on a class. The following metadata is provided to create a module.

**imports:** Dependency on *BrowserModule* is specified with *imports*. Notice it's an array. Multiple modules' dependency could be specified with imports.

**declarations:** All components associated with the current module could be specified as values in *declarations* array.

**bootstrap:** Each Angular application needs to have a root module which will have a root component that is rendered on index.html. For the current module, MyAppComponent (imported from a relative path) is the first component to load.

## o6. Service

Create a service in an Angular application for any reusable piece of code. It could be accessing a server side service API, providing configuration information etc.

To create a service, import and decorate a class with *@injectable* (TypeScript) from *@angular/core module*. This allows angular to create an instance and inject the given service in a component or another service.

Consider the following hypothetical sample. It creates a *TimeService* for providing date time values in a consistent format across the application. The *getTime()* function could be called in a component within the application.

```
import { Injectable } from '@angular/core';

@Injectable()
export class TimeService {
  constructor() { }
  getTime(){
    return `${new Date().getHours()} : ${new Date().getMinutes()} : ${new Date().getSeconds()}`;
  }
}
```

### 6.1 Dependency Injection (DI)

In the service section, we have seen creating a service by decorating it with *Injectable()* function. It helps angular injector create an object of the service. It is required only if the service has other dependencies injected. However, it's a good practice to add *Injectable()* decorator on all services (even the ones without dependencies) to keep the code consistent.

With DI, Angular creates an instance of the service, which offloads object creation, allows implementing singleton easily, and makes the code conducive for unit testing.

### 6.2 Provide a service

To inject the service in a component, specify the service in a list of a providers. It could be done at the module level (to be accessible largely) or at component level (to limit the scope). The service is instantiated by Angular at this point.

Here's a component sample where the injected service instance is available to the component and all its child components.

```
import { Component } from '@angular/core';
import { TimeService } from '../time.service';
@Component({
  selector: 'app-root',
  providers: [TimeService],
  template: `<div>Date: {{timeValue}}</div>`,
})
export class SampleComponent {
  // Component definition
}
```

To complete the DI process, specify the service as a parameter property in the constructor.

```
export class SampleComponent {
  timeValue: string = this.time.getTime(); // Template shows

  constructor(private time: TimeService){
  }
}
```

Refer to the TimeService implementation below,

```
import { Injectable } from '@angular/core';
@Injectable()
export class TimeService {
  constructor() { }
  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}`;
  }
}
```

### 6.3 Provide a service at module level

When the TimeService is provided at module level (instead of the component level), the service instance is available across the module (and application). It works like a Singleton across the application.

```
@NgModule({
  declarations: [
    AppComponent,
    SampleComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [TimeService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

### 6.4 Alternative syntax to provide a service

In the above sample, provider statement is a short cut to the following,

```
// instead of providers: [TimeService] you may use the following,
providers: [{provide: TimeService, useClass: TimeService}]
```

It allows using a specialized or alternative implementation of the class (TimeService in this example). The new class could be a derived class or the one with similar function signatures to the original class.

```
providers: [{provide: TimeService, useClass: AlternateTimeService}]
```

For the sake of an example, the *AlternateTimeService* provides date and time value. The original *TimeService* provided just the date value.

```
@Injectable()
export class AlternateTimeService extends TimeService {
  constructor() {
    super();
  }

  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}
            ${dateObj.getHours()}:${dateObj.getMinutes()}`;
  }
}
```

**Note:** Angular injector creates a new instance of the service when the following statement is used to provide a service:

```
[{provide: TimeService, useClass: AlternateTimeService}]
```

To rather use an existing instance of the service, use `useExisting` instead of `useClass`:

```
providers: [AlternateTimeService, {provide: TimeService, useExisting:
AlternateTimeService}]
```

## 6.5 Provide an Interface and its implementation

It might be a good idea to provide an interface and implementation as a class or a value. Consider the following interface.

```
interface Time{
  getTime(): string
}
export default Time;
```

It could be implemented by *TimeService*.

```
@Injectable()
export class TimeService implements Time {
  constructor() { }

  getTime(){
    let dateObj: Date = new Date();
    return `${dateObj.getDay()}/${dateObj.getMonth()}/${dateObj.getFullYear()}`;
  }
}
```

At this point, we can't implement *Time* interface and *TimeService* class like the above sample.

The following piece of code does **not** work, because a TypeScript interface doesn't compile to any equivalent in JavaScript.

```
providers: [{provide: Time, useClass: TimeService}] // Wrong
```

To make this work, import Inject (Decorator) and InjectionToken from @angular/core.

```
// create an object of InjectionToken that confines to interface Time
let Time_Service = new InjectionToken<Time>('Time_Service');

// Provide the injector token with interface implementation.
providers: [{provide: Time_Service, useClass: TimeService}]

// inject the token with @inject decorator
constructor(@Inject(Time_Service) ts,) {
  this.timeService = ts;
}

// We can now use this.timeService.getTime().
```

## 6.6 Provide a value

We may not always need a class to be provided as a service.

The following syntax allows for providing a JSON object. Notice JSON object has the function `getTime()`, which could be used in components.

```
providers: [{provide: TimeService, useValue: {
  getTime: () => `${dateObj.getDay()} - ${dateObj.getMonth()} - ${dateObj.getFullYear()}`
}}]
```

**Note:** For an implementation similar to the section “Provide an Interface and its implementation”, provide with `useValue` instead of `useClass`. The rest of the implementation stays the same.

```
providers: [provide: Time_Service, useValue: {
  getTime: () => 'A date value'
}]]
```

## 07. Directives

From Angular 2 onwards, directives are broadly categorized as following:

1. Components - Includes template. They are the primary building blocks of an Angular application. Refer to the Component section in the article.
2. Structural directives – A directive for managing layout. It is added as an attribute on the element and controls flow in DOM. Example NgFor, NgIf etc.
3. Attribute directives – Adds dynamic behavior and styling to an element in the template. Example NgStyle.

### 7.1 Ng-if... else

A structural directive to show a template conditionally. Consider the following sample. The `ngIf` directive guards against showing half a string *Time: [with no value]* when the title is empty. The *else* condition shows a template when no value is set for the *title*.



Notice the syntax here, the directive is prefixed with an asterisk.

```
<div *ngIf="title; else noTitle">
  Time: {{title}}
</div>
```

```
<ng-template #noTitle> Click on the button to see time. </ng-template>
```

As and when the *title* value is available, *Time: [value]* is shown. The *#noTitle* template hides, as it doesn't run *else*.

## 7.2 Ng-Template

Ng-Template is a structural directive that doesn't show by default. It helps group content under an element. By default, the template renders as a HTML comment.

The content is shown conditionally.

```
<div *ngIf="isTrue; then tplWhenTrue else tplWhenFalse"></div>
<ng-template #tplWhenTrue >I show-up when isTrue is true. </ng-template>
<ng-template #tplWhenFalse > I show-up when isTrue is false </ng-template>
```

**Note:** Instead of ng-if...else (as in *ng-if...else* section), for better code readability, we can use *if...then...else*. Here, the element shown when condition is true, is also moved in a template.

## 7.3 Ng-Container

Ng-container is another directive/component to group HTML elements.

We may group elements with a tag like **div** or **span**. However, in many applications, there could be default style applied on these elements. To be more predictable, Ng-Container is preferred. It groups elements, but doesn't render itself as a HTML tag.

Consider following sample,

```
// Consider value of title is Angular
Welcome <div *ngIf="title">to <i>the</i> {{title}} world.</div>
```

It is rendered as

```

| Welcome
| to the Angular world.
```

The resultant HTML is as below,

```

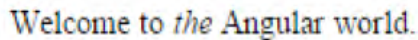
Welcome "
<!--bindings=
  "ng-reflect-ng-if": "Angular"
-->
<div _ngcontent-c0>
  "to "
  <i _ngcontent-c0>the</i>
  " Angular world."
</div>
<hr _ngcontent-c0>
```

It didn't render in one line due to the div. We may change the behavior with CSS. However, we may have a

styling applied for `div` by default. That might result in an unexpected styling to the string within the `div`.

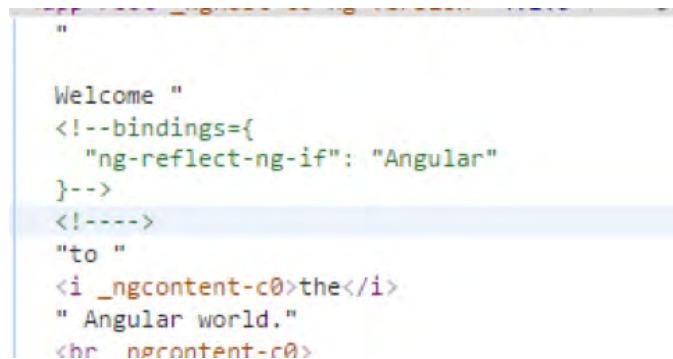
Now instead, consider using `ng-container`. Refer to the snippet below.

```
Welcome <ng-container *ngIf="title">to <i>the</i> {{title}} world.</ng-container>
The result on the webpage is as following
```



Welcome to *the* Angular world.

The resultant HTML is as below. Notice `ng-container` didn't show up as an element.



```
"
Welcome "
<!--bindings=
  "ng-reflect-ng-if": "Angular"
-->
<!-->
"to "
<i _ngcontent-c0>the</i>
" Angular world."
</_ngcontent-c0>
```

## 7.4 NgSwitch and NgSwitchCase

We can use **switch-case** statement in Angular templates. It's similar to `switch..case` statement in JavaScript.

Consider the following snippet. `isMetric` is a variable on the component. If its value is true, it will show Degree Celsius as the label, else it will show Fahrenheit.

**Notice `ngSwitch` is an attribute directive and `ngSwitchCase` is a structural directive.**

```
<div [ngSwitch]="isMetric">
  <div *ngSwitchCase="true">Degree Celsius</div>
  <div *ngSwitchCase="false">Fahrenheit</div>
</div>
```

Please note, we may use `ngSwitchDefault` directive to show a default element when none of the values in the `switch...case` are true.

Considering `isMetric` is a boolean variable, the following code will result in the same output as the previous snippet.

```
<div [ngSwitch]="isMetric">
  <div *ngSwitchCase="true">Degree Celsius</div>
  <div *ngSwitchDefault>Fahrenheit</div>
</div>
```

## 7.5 Input decorator function

A class variable could be configured as an input to the directive. The value will be provided by component using the directive.

Consider the following code snippet. It is a component that shows login fields. (A component is a type of directive).

The Component invoking login component could set *showRegister* to true, resulting in showing a register button.

To make a class variable input, annotate it with the Input decorator function.

```
import Input()
import { Component, OnInit, Input } from '@angular/core';
```

Annotate with the decorator function:

```
@Input() showRegister: boolean;
```

Use the input value in the component which is in a template in this example.

```
<div>
  <input type="text" placeholder="User Id" />
  <input type="password" placeholder="Password" />
  <span *ngIf="showRegister"><button>Register</button></span>
  <button>Go</button>
</div>
```

While using the component, provide the input:

```
<login shouldShowRegister="true"></login>
```

To use binding instead of providing the value directly, use the following syntax:

```
<app-login [shouldShowRegister]="isRegisterVisible"></app-login>
```

We could provide a different name to the attribute than that of the variable. Consider the following snippet:

```
@Input("should-show-register") showRegister: boolean;
```

Now, use the attribute *should-show-register* instead of the variable name *showRegister*.

```
<app-login should-show-register="true"></app-login>
```

## 7.6 Output decorator function

Events emitted by a directive are output to the component (or a directive) using the given directive. In the login example, on clicking login, an event could be emitted with user id and password.

Consider the following snippet. Import *Output* decorator function and *EventEmitter*,

```
import { Component, OnInit, Input, Output, EventEmitter } from '@angular/core';
```

Declare an Output event of type *EventEmitter*

```
@Output() onLogin: EventEmitter<{userId: string, password: string}>;
```

Notice the generic type (anonymous) on *EventEmitter* for *onLogin*. It is expected to emit an object with user id and password.

Next, initialize the object.

```
constructor() {
  this.onLogin = new EventEmitter();
}
```

The component emits the event. In the sample, when user clicks on login, it emits the event with a next function.

Here's the template:

```
<button (click)="loginClicked(userId.value, password.value)">Go</button>
```

..and the Event handler:

```
loginClicked(userId, password){
  this.onLogin.next({userId, password});
}
```

While using the component, specify login handler for the onLogin output event.

```
<app-login (onLogin)="loginHandler($event)"></app-login>
```

Login handler receives user id and password from the login component

```
loginHandler(event){
  console.log(event);
  // Perform login action.
}
```

```
// Output: Object {userId: "sampleUser", password: "samplePassword"}
```

Similar to the input decorator, output decorator could specify an event name to be exposed for consuming component/directive. It need not be the variable name. Consider the following snippet.

```
@Output("login") onLogin: EventEmitter<{userId: string, password: string}>;
```

..while using the component:

```
<app-login (login)="loginHandler($event)"></app-login>
```

## o8. Change Detection Strategy

Angular propagates changes top-down, from parent components to child components.

Each component in Angular has an equivalent change detection class. As the component model is updated, it compares previous and new values. Then changes to the model are updated in the DOM.

For component inputs like number and string, the variables are immutable. As the value changes and the change detection class flags the change, the DOM is updated.

For object types, the comparison could be one of the following,

**Shallow Comparison:** Compare object references. If one or more fields in the object are updated, object reference doesn't change. Shallow check will not notice the change. However, this approach works best for immutable object types. That is, objects cannot be modified. Changes need to be handled by creating a new instance of the object.

**Deep Comparison:** Iterate through each field on the object and compare with previous value. This identifies change in the mutable object as well. That is, if one of the field on the object is updated, the change is noticed.

### *8.1 Change detection strategy for a component*

On a component, we can annotate with one of the two change detection strategies.

#### **ChangeDetectionStrategy.Default:**

As the name indicates, it's the default strategy when nothing is explicitly annotated on a component.

With this strategy, if the component accepts an object type as an input, it performs deep comparison every time there is a change. That is, if one of the fields have changed, it will iterate through all the fields and identify the change. It will then update the DOM.

Consider the following sample:

```
import { Component, OnInit, Input, ChangeDetectionStrategy } from '@angular/core';

@Component({
  selector: 'app-child',
  template: ' <h2>{{values.title}}</h2> <h2>{{values.description}}</h2>',
  styleUrls: ['./child.component.css'],
  changeDetection: ChangeDetectionStrategy.Default
})
export class ChildComponent implements OnInit {

  // Input is an object type.
  @Input() values: {
    title: string;
    description: string;
  }
  constructor() { }

  ngOnInit() {}
}
```

Notice, change detection strategy is mentioned as **ChangeDetectionStrategy.Default**, it is the value set by default in any component. *Input* to the component is an object type with two fields *title* and *description*.

Here's the snippet from parent component template.

```
<app-child [values]="values"></app-child>
```

An event handler in the parent component is mentioned below. This is triggered, possibly by a user action or other events.

```
updateValues(param1, param2){
  this.values.title = param1;
  this.values.description = param2;
}
```

Notice, we are updating the values object. The Object reference doesn't change. However the child component updates DOM as the strategy performs deep comparison.

### **ChangeDetectionStrategy.OnPush:**

The default strategy is effective for identifying changes. However, it's not performant as it has to loop through a complete object to identify change. For better performance with change detection, consider using *OnPush* strategy. It performs shallow comparison of input object with the previous object. That is, it compares only the object reference.

The code snippet we just saw will not work with *OnPush* strategy. In the sample, the reference doesn't change as we modify values on the object. The Child component will not update the DOM.

When we change strategy to *OnPush* on a component, the input object needs to be immutable. We should create a new instance of input object, every time there is a change. This changes object reference and hence the comparison identifies the change.

Consider the following snippet for handling change event in the parent.

```
updateValues(param1, param2){
  this.values = { // create a new object for each change.
    title: param1,
    description: param2
  }
}
```

**Note:** Even though the above solution for *updateValues* event handler might work okay with the *OnPush* strategy, it's advisable to use [immutable.js](#) implementation for enforcing immutability with objects or observable objects using RxJS or any other observable library.

## **09. Transclusion in Angular**

In the above two sections (input decorator and output decorator), we have seen how to use component attributes for input and output.

How about accessing content within the element, between begin and end tags?

AngularJS 1.x had transclusion in directives that allowed content within the element to render as part of the directive. For example, an element `<blue></blue>` can have elements, expressions and text within the element. The component will apply blue background color and show the content.

```
<blue>sample text</blue>
```

In the blue component's template, use `ng-content` to access content within the element.

Consider the following. The CSS class blue applies styling.

```
<div class="blue">
  <ng-content>
</ng-content></div>
```

*ng-content* shows *sample text* from above example. It may contain more elements, data binding expressions etc.

## 10. Using observables in Angular templates

Observable data by definition, may not be available while rendering the template. The `*ngIf` directive could be used to conditionally render a section. Consider the following sample.

```
<div *ngIf="asyncData | async; else loading; let title">
  Title: {{title}}
</div>
<ng-template #loading> Loading... </ng-template>
```

Notice the `async` pipe. The sample above checks for `asyncData` observable to return with data. When observable doesn't have data, it renders the template *loading*. When the observable returns data, the value is set on a variable `title`. The `async` pipe works the same way with promises as well.

### *NgFor*

Use `*ngFor` directive to iterate through an array or an observable. The following code iterates through `colors` array and each item in the array is referred to as a new variable `color`.

```
<ul *ngFor="let color of colors">
  <li>{{color}}</li>
</ul>
/* component declares array as colors= ["red", "blue", "green", "yellow",
"violet"];*/
```

**Note:** Use *async pipe* if `colors` is an observable or a promise.

## 11. Strict Null Check

TypeScript 4 introduced strict null check for better type checking. TypeScript (& JavaScript) have special types namely *null* and *undefined*.

With strict type check enabled for an Angular application, null or undefined cannot be assigned to a variable unless they are of that type. Consider the following sample:

```
var firstName:string, lastName:string ;
//returns an error, Type 'null' is not assignable to type 'string'.
firstName=null;
// returns an error, Type 'undefined' is not assignable to type 'string'.
lastName=undefined;
```

Now explicitly specify null and undefined types. Revisit variable declaration as following:

```
var firstName:string|null, lastName:string|undefined ;
//This will work
firstName=null;
lastName=undefined;
```

By default, `strictNullCheck` is disabled. To enable `strictNullCheck` edit `tsconfig.json`. Add `"strictNullChecks": true` in `compilerOptions`

```

{
  "compileOnSave": false,
  "compilerOptions": {
    "outDir": "./dist/out-tsc",
    "baseUrl": "src",
    "sourceMap": true,
    "declaration": false,
    "moduleResolution": "node",
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "strictNullChecks": true,
    "target": "es5",
    "typeRoots": [
      "node_modules/@types"
    ],
  },
}

```

Figure 2: tsconfig.json

## 12. HTTP API calls and Observables

Use the built-in Http service from '@angular/http' module to make server side API calls. Inject the Http service into a component or another service. Consider this snippet which makes a GET call to a Wikipedia URL. Here http is an object of Http in @angular/http

```

this.http
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")

```

The get() function returns an observable. It helps return data asynchronously to a callback function. Unlike promise, an observable supports a stream of data. It's not closed as soon as data is returned. Moreover it can be returned till the observable is explicitly closed.

Observables support multiple operators or chaining of operators. Consider the map operator below which extracts JSON out of the http response.

```

this.http
  .get("http://localhost:3000/dino")
  .map((response) => response.json())

```

**Note:** Shown here is one of the ways to import map operator.

```

import 'rxjs/add/operator/map'; // Preferred way to import

```

Otherwise, we may use the following import statement for the entire rxjs library.

```

import 'rxjs';

```

However, it still returns an observable. An observable has a subscribe function, which accepts three callbacks.

- Success callback that has data input
- Error callback that has error input
- Complete callback, which is called while finishing with the observable.



Consider the following snippet,

```
this.http
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
  .map((response) => response.json())
  .subscribe((data) => console.log(data), // success
    (error) => console.error(error), // failure
    () => console.info("done")); // done
```



```
Object {bruathkayosaurus: Object, lambeosaurus: Object, linhenykus: Object,
  pterodactyl: Object, stegosaurus: Object...}
  ▶ bruhathkayosaurus: Object
  ▶ lambeosaurus: Object
  ▶ linhenykus: Object
  ▶ pterodactyl: Object
  ▶ stegosaurus: Object
  ▶ triceratops: Object
  ▶ __proto__: Object
done
```

Figure 3: console output of observable.

You may set the observable result to a class/component variable and display it in the template. A better option would be to use async pipe in the template.

Here's a sample:

```
dino: Observable<any>; // class level variable declaration

// in the given function, where the API call run set returned observable to the //
class variable

this.dino = this.http // Set returned observable to a class variable
  .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
  .map((response) => response.json());
```

In the template use async pipe. When the dino object is available, use fields on the object.

```
<div>{{ (dino | async)?.bruathkayosaurus.appeared }}</div>
```

## 13. Take me back to Promises

We may use an RxJS operator to convert the *observable* to a *promise*. If an API currently returns a promise, for backward compatibility, it's a useful operator.

Import *toPromise* operator:

```
import 'rxjs/add/operator/toPromise';
```

Use *toPromise* on an observable:

```
getData(): Promise<any>{
  return this.http
    .get("https://dinosaur-facts.firebaseio.com/dinosaurs.json")
    .map((response) => response.json())
    .toPromise();
}
```

Use data returned on the success callback. Set it to a class variable to be used in the template.

```
this.getData()
  .then((data) => this.dino = data)
  .catch((error) => console.error(error));
```

Review the following template:

```
<div *ngIf="dino">
  <div>Bruhathkayosaurus appeared {{ dino.bruhathkayosaurus.appeared }} years ago
</div>
```

## 14. Router

Routing makes it possible to build an SPA (Single Page Application). Using the router configuration, components are mapped to a URL.

To get started with an Angular application that supports routing using Angular CLI, run the following command.

```
ng new sample-application --routing
```

To add a module with routing to an existing application using Angular CLI, run the following command.

```
ng g module my-router-module --routing
```

### 14.1 Router Outlet

Components at the route (URL) are rendered below router outlet. In general, RouterOutlet is placed in the root component for routing to work.

```
<router-outlet></router-outlet>
```

### 14.2 Route configuration

Create a routes object of type Routes ( in@angular/router package) which is an array of routes JSON objects. Each object may have one or more of the following fields.

**path:** Specifies path to match

**component:** At the given path, the given component will load below router-outlet

**redirectTo:** Redirects to the given path, when path matches. For example, it could redirect to home, when no path is provided.

**pathMatch:** It allows configuring path match strategy. When the given value is *full*, the complete path needs to match. Whereas *prefix* allows matching initial string. Prefix is the default value.

Consider the following route configuration for sample:

```
const routes: Routes = [
  {
    path: 'home',
    component: Sample1Component,
  },
  {
    path: 'second2',
```

```

    component: Sample2Component
  },
  {
    path: '',
    redirectTo: '/home',
    pathMatch: 'full'
  }
];

```

### 14.3 Child Routes

For configuring child routes, use children within the route object. The child component shows-up at the router-outlet in the given component.

The following sample renders Sample1Component. It has a router-outlet in the template. Sample2Component renders below it.

```

const routes: Routes = [
  {
    path: 'home',
    component: Sample1Component,
    children: [
      {
        path: 'second',
        component: Sample2Component
      }
    ]
  }
]... // rest of the configuration.

// Template for Sample1Component

<div>
  sample-1 works!
  <router-outlet></router-outlet> <!--Sample2Component renders below it -->
</div>

```

### 14.4 Params

Data can be exchanged between routes using URL parameters.

Configure variable in the route: In the sample below, the *details* route expects an id as a parameter in the URL. Example <http://sample.com/details/10>

```

{
  path: details/:id,
  component: DetailsComponent
}

```

Read value from the URL: Import ActivatedRoute from @angular/router. Inject ActivatedRoute and access params. It's an observable to read value from the URL as seen in the sample:

```

// inject activatedRoute
constructor(private activeRoute: ActivatedRoute) { }

// Read value from the observable
this.activeRoute

```

```
.params  
.subscribe((data) => console.log(data[id]));
```

## Conclusion

Angular as a JavaScript framework has been evolving fast. From being an MV\* framework in AngularJS 1.x, it has become a framework that approaches UI development with reusable components.

With Angular 2 and above (current version being 4.x) performance has improved and new features are continuously added. The newer versions are referred to as just Angular, instead of qualifying it with a version number.

Angular with TypeScript has multitude of features that make development in JavaScript faster and easier. The article described some of the features to begin developing with Angular.

Happy coding with the super heroic framework, Angular!

## *References*

Angular documentation – <https://angular.io/docs>

Change Detection Strategy with Angular - [https://angular-2-training-book.rangle.io/handout/change-detection/change\\_detector\\_classes.html](https://angular-2-training-book.rangle.io/handout/change-detection/change_detector_classes.html)

Dinosaur data - <https://dinosaur-facts.firebaseio.com/dinosaurs.json> (Firebase API for Dinosaur data)