Angular 7

Trusted partner for your **Digital Journey**

Atos | Syntel

Version Control and Revision History

	Name	Date
Prepared By	Pradeep Chinchole	21-Feb-2019
Reviewed By	Liji Shynu	21-Feb-2019
Approved By	Nisha Mendonsa	26-Feb-2019

Version No.	Date	Section Affected	Highlight of Changes
1.0.0	26-Feb-2019	All	Original Version

Agenda

- Angular JS vs Angular 6
- Introduction to Angular 6
- Angular Architecture
- Angular Directives
- Angular Pipes
- Angular Services
- Angular Forms
- Angular Routing
- Angular Material
- Angular Animation
- Angular cdk

Angular JS vs Angular

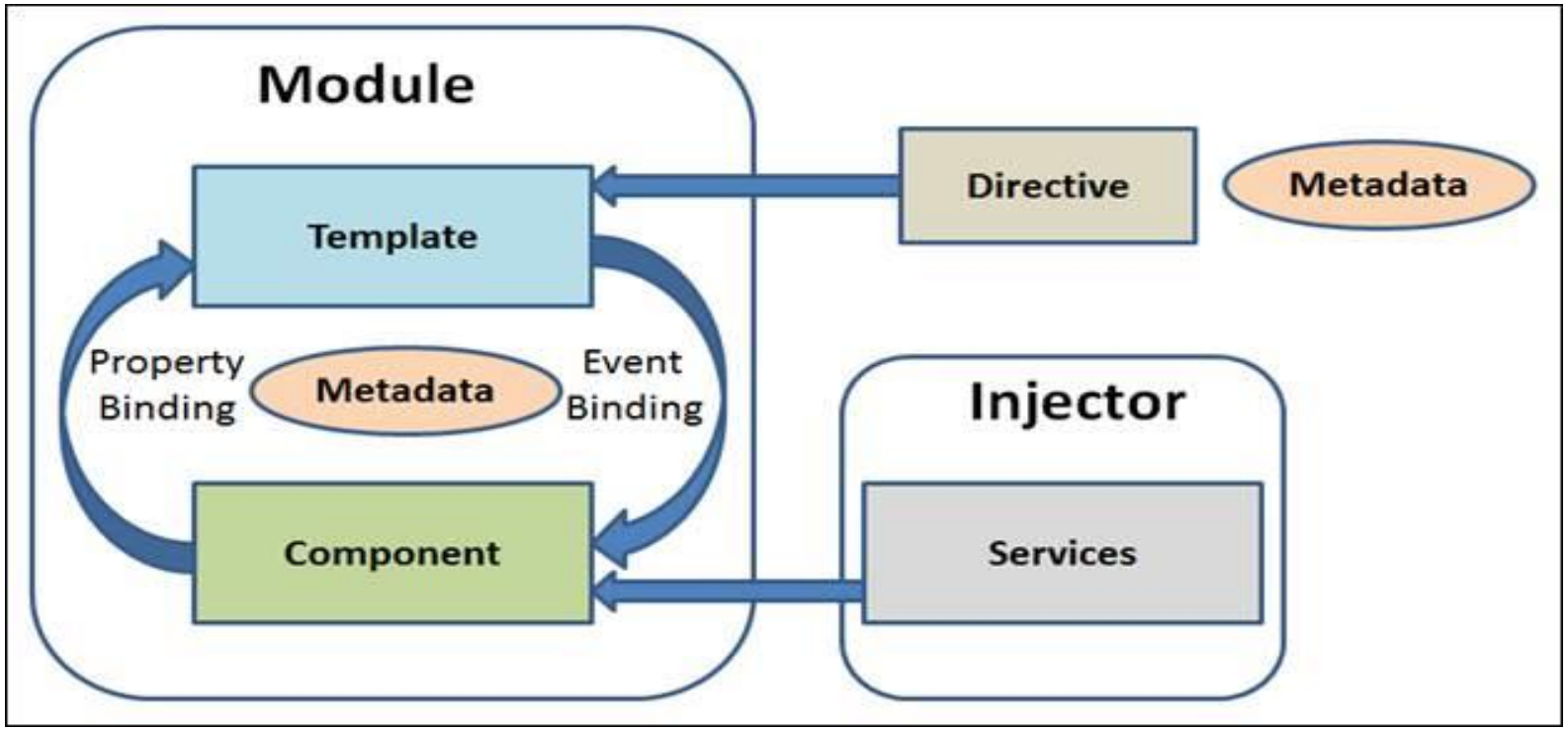
Angular JS (1.6)	Angular (5)
1. It's not developed by keeping mobile development mind	1.Anular is developed for mobile development
2. Base language is Java Script	2.Base language Type Script (Object Oriented Version of Java Script)
3.Controller	3.Components and Templates
4.Filters	4.Pipes
5.Routing is function based	5.Routing is JSON based
6.Directives :ng-if ng-repeat	6.Directives : *ngIf *ngFor
7.Angular JS development is easy	7.Angular development is complicated as Type Script is not a browser understandable code.

What is Angular?

- Angular is a platform that makes it easy to build applications with the web.
- Angular combines declarative templates, dependency injection, end to end tooling, and integrated best practices to solve development challenges.
- Angular empowers developers to build applications that live on the web, mobile, or the desktop

Angular Architecture

Angular Architecture



The main building blocks of Angular

- Module
- Component
- Metadata
- Template
- Data Binding
- Pipes
- Service
- Directive
- Dependency Injection

Module

- Module is the block of code which is designed to perform a single task.
- We can export the module in form of class.
- Angular applications have one or more modules.
- Every Angular application must have at least one module.
- If Angular application contains only one module, it is referring as root module.
- Every Angular application has one root module and many more featured modules.

Module

- Angular module is a class which is decorated with @NgModule.
- NgModule takes a single metadata object and its properties describe the module.
- exports - It is the subset of declarations which would be used in the component template of other module.
- imports - imports other modules
- providers - It is a creator of services. They can be accessible in all the parts of the application.
- bootstrap - The root module has to set the bootstrap property. It is used to host all other views.
- declarations - It declare the view class that belong to current module. There are three type of view classes supported by Angular components, directives, and pipes.

Module Example

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

Component

- The component is class with the template that deals with the View of application and it's containing the core logic for the page.
- We can compare it with the Controller in Angular 1.x.
- We need to write the application logic inside the class which is used by the View.
- The component class interacts with the View through Methods and Properties of API. Angular creates and updates the required component and destroys the unused component as the user moves through an application.

Component Example

```
import { Component } from '@angular/core';
@Component({
selector: 'test-app',
template: '<h1>This is my First Angular  Application</h1>' +
'<br/>' +
'<input #txtName type = "text" (keyup)="0" />' +
'<br/> ' +
'<p>You have Enter: {{txtName.value}}</p>'
})
export class AppComponent {
  }
```

Metadata

- Metadata is the way of defining the processing of a class.
- In TypeScript, we can define metadata by using decorator.
- For example, if we define any component in Angular application, we need to tell Angular that this is the component, by using metadata of the class (using @Component decorator).

Metadata Example

```
@Component({
selector: 'test-app',
template: '<h1>This is my First Angular 2 Application</h1>' +
'<br/>' +
'<input #txtName type = "text" (keyup)="0" />' +
'<br/> ' +
'<p>You have Enter: {{txtName.value}}</p>'
})
```

The decorator @Component is used to configure the object to create the component and its View. The selector is creating the instance of the component. In the above example code, if Angular finds <test-app> tag in HTML, it replaces it with the template defined in component.

Template

- The template is the component View that tells Angular how to display the component.
- It looks like normal HTML.
- Templates include data bindings as well as other components and directives

Data Binding

- It Enables data to flow from the component to template and vice-versa
- There are four type of binding supported by Angular application,
- Interpolation - It displays the component value within the HTML tags which is also referred as Expression in Angular 1.x.
- Property Binding - It passes the value of property from the parent to the property of the child.
- Event Binding - It fires the event when we click on the components method name.
- Two-way Binding - It is an important form that combines event and property binding in single notation by using ngModel directive.

Data Binding Example

app.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({ selector: 'app-employee-form',
                templateUrl: 'app.component.html',
               styleUrls: []   })
export class AppComponent implements OnInit {
itemCount: number = 1;
btnText:string="Add an Item";
message:string="Hello Students";
goals=[];
goal='Be a good human being';
constructor() { }  ngOnInit() {}
addItem() {
    this.goals.push(this.goal);    this.goal= '';  this.itemCount = this.goals.length;
  }}
```

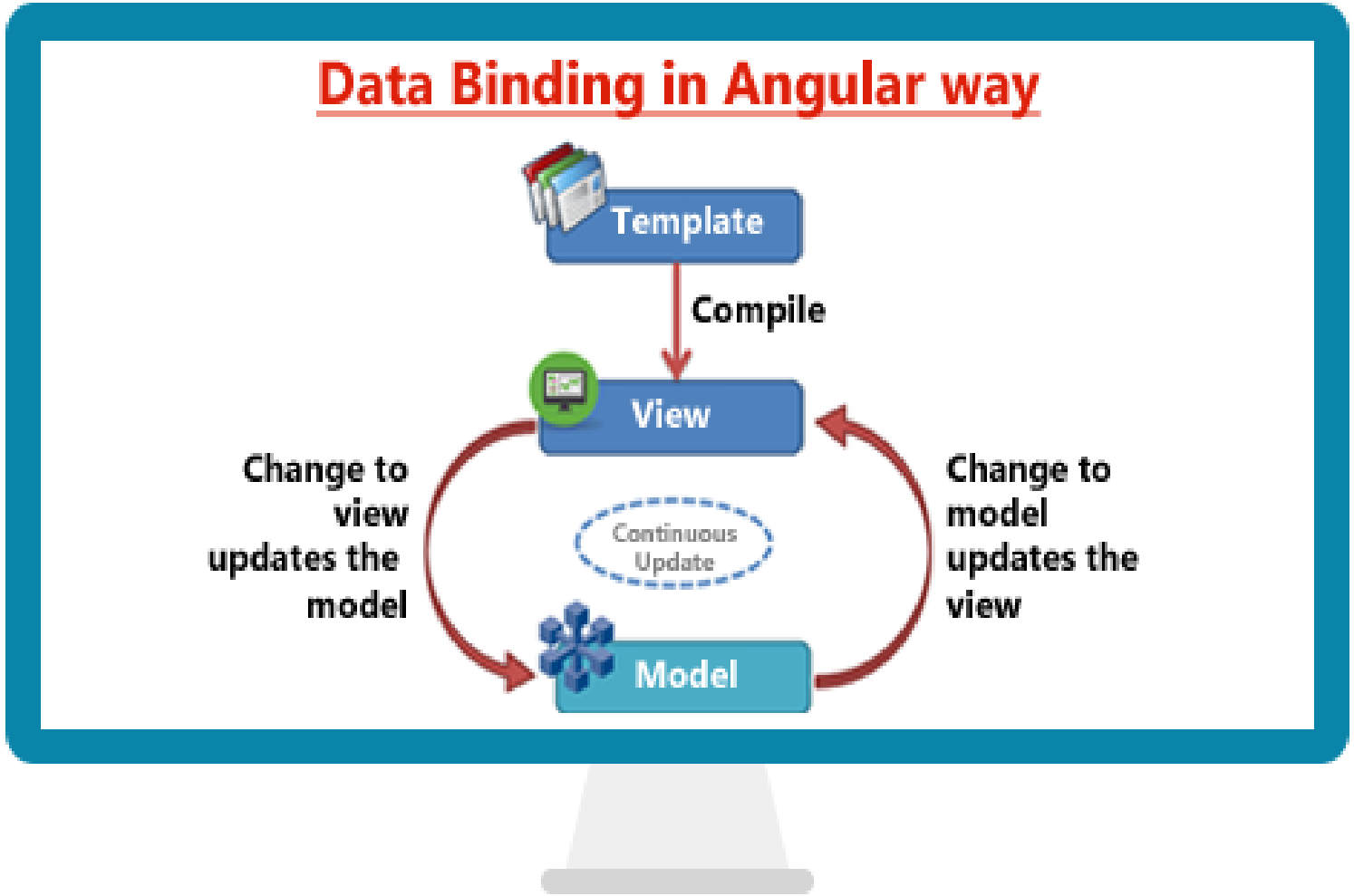
Data Binding Example

- app.component.html

```
<!-- Interpolation Example: -->
        <div> Item Count : ({{ itemCount }})</div>
<!-- Property Binding  : -->
        <input type="submit" class="btn" [value]="btnText">
        <input type="submit" class="btn" value="{{ btnText }}">
<!– One way data binding  : -->
        <input  name="message" placeholder="Message.."
[ngModel]="message">
        <span>{{ message }}</span><br>

<!– Two way data binding  : -->
        <input  name="message" placeholder="Message.."
[(ngModel)]="message">
        <span>{{ message }}</span><br>
```

Two way Data Binding



Angular Pipes

Pipes

- Angular pipes are used to format sort or search the data
- Below are the Angular built-in pipes
  - lowercase
  - Uppercase
  - titlecase
  - number
  - percent
  - currency
  - date
  - json
  - slice
  - async

Angular Custom Pipes

- Custom Pipes (previously filter in AngularJS) allow us to essentially create a pure function, which accepts an input and returns a different output via some form of transformation.

In Angular to create a custom pipe we need to write a class decorated with @Pipe decorator and implements PipeTransform interface and implement transform method

reverse.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({name: 'reverse'})
export class Reverse implements PipeTransform {
transform(value: string): string {
let newStr: string = "";
for (var i = value.length - 1; i >= 0; i--) {
newStr += value.charAt(i);}
return newStr;
}}
```

Angular Custom Pipes

- **app.module.ts**

```
import { NgModule } from '@angular/core';
import { AppComponent } from './app.component';
import {Reverse} from "./reverse.pipe.ts";
@NgModule({
 declarations: [    AppComponent,Reverse],
 imports: [    BrowserModule,],
 providers: [],
 bootstrap: [AppComponent]})
export class AppModule { }
```

Angular Custom Pipes

- **app.component.ts**

```
import { Component, OnInit } from '@angular/core';
@Component({
selector: 'app-employee-form',
 template: ` Reverse of  {{user}}     is {{user | reverse}}',
 styleUrls: []})
export class AppComponent implements OnInit {
user="Pradeep Chinchole";
ngOnInit(){  }
}
}
```

Angular Services

Service

- It is Type Script class decorated with @Injectable which is used to perform a specific task.
- It includes the function, values, or any other feature required by the application.
- Typical examples of services are logging service, data service, message service etc. There is no base class for service.

Angular Service Example

app.component.ts

```
import { Injectable } from '@angular/core';
@Injectable()
export class LoggingService {
}
```

Angular Service Example

- **app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { LoggingService } from './logging.service;
@NgModule({
  declarations: [AppComponent],
  imports: [ BrowserModule, FormsModule,ReactiveFormsModule //modules],
  providers: [LoggingService], //services
  bootstrap: [AppComponent] //Components
})export class AppModule { }
```

Angular Directives

Directive

- Directives are one of the most important components of AngularJS application.
- They are extended HTML attributes.
- These are the markers on the DOM element which provides some special behavior to DOM elements and tell AngularJS's HTML compiler to attach.
- It is class with directive metadata.
- To create directive, we have to apply @Directive decorator on attached metadata to the class.
- There are three types of directives
  1. Component Directive
  2. Structural Directive
  3. Attribute Directive

Component Directives

- Component Directive form the main class having details of how the component should be processed, instantiated and used at runtime.

  Ex.

  <router-outlet>

  <ng-template>

  <app-root>

Attribute Directives

- Attribute directives deal with changing the look and behavior of the dom element. You can create your own directives as shown below.

  Ex.

  [ngModel]

  [ngStyle]

  [ngClass]

  [ngSwitch]

  [ngForm]

Structural Directives

- Structural directives are responsible for HTML layout. They shape or reshape the DOM's *structure*, typically by adding, removing, or manipulating elements.
  - **Ex**.

  *ngIf

  *ngFor

  *ngSwitchCase

  *ngSwitchDefault

Dependency Injection

Dependency Injection

- Dependency Injection is a software design pattern in which objects are passed as dependencies.
- It helps us remove the hard coded dependencies, and makes dependencies configurable.
- Using Dependency Injection, we can make components maintainable, reusable, and testable.
- Point to remember about Dependency Injection,
- To use DI for a service, we register it as a provider in one of two ways:
  - when bootstrapping the application,
  - or in the component metadata

Dependency Injection Example

```
import { Component } from '@angular/core';
import { LoggingService } from ’./logging.service.ts';
@Component({
selector: 'test-app',
template: `Hello World' ,
providers:[LoggingService]
})
export class AppComponent {
constructor(private ls:LoggingService)   //Dependecny Injection
{}
}
```

Angular HttpClient

- Most front-end applications communicate with backend services over the HTTP protocol.
- Modern browsers support two different APIs for making HTTP requests: the XMLHttpRequest interface and the fetch() API.
- The HttpClient in @angular/common/http offers a simplified client HTTP API for Angular applications that rests on the XMLHttpRequest interface exposed by browsers.
- Additional benefits of HttpClient include testability features, typed request and response objects, request and response interception, Observable apis, and streamlined error handling.

Angular Http Client Example

- **post.service.ts**

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
@Injectable()
export class PostsService {
private baseURL="https://jsonplaceholder.typicode.com/posts"
constructor(private http: HttpClient) {}
 getAllPosts():Observable<any>{
 return this.http.get(this.baseURL)
 }
 getAllPostsByUserId(userId:number):Observable<any>{
 return this.http.get(this.baseURL+"?userId="+userId)
 }
}
```

Angular HttpClient Example

- **app.module.ts**

```
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';
import { AppComponent } from './app.component';
import {PostService} from "./post.service.ts";
@NgModule({
 declarations: [    AppComponent,  ],
 imports: [    BrowserModule,    HttpModule,    HttpClientModule ],
 providers: [PostsService],
 bootstrap: [AppComponent]})
export class AppModule { }
```

Angular Form Validation

Angular Forms

- Forms are almost always present in any website or application. Forms can be used to perform countless data-entry tasks such as: authentication, order submission or a profile creation.
- Angular form validation Improve overall data quality by validating user input for accuracy and completeness.
- Building easy-to-use forms requires design and user experience skills, as well as a framework with support for two-way data binding, change tracking, validation, and error handling such as Angular.
- When it comes to form-building, Angular offers two technologies: Reactive forms and Template driven

Angular Forms

Template Driven Forms	Reactive Forms
1.Suitable for simple scenarios and fails for complex scenarios	1.More flexible, but needs a lot of practice
2.Similar to AngularJS	2.Handles any complex scenarios
3.Two way data binding(using [(NgModel)] syntax)	3.No data binding is done (immutable data model preferred by most developers)
4.Minimal component code	4.More component code and less HTML markup
5.Automatic track of the form and its data (handled by Angular)	5.Reactive transformations can be made possible such as A)Handling a event based on a debounce time, B)Handling events when the components are distinct until changed C)Adding elements dynamically
6.Unit testing is another challenge	6.Easier unit testing

Angular Template Driven Form Example

- **employee-form.component.ts**

```
import { Component, OnInit } from '@angular/core';
@Component({
selector: 'app-employee-form',
 templateUrl: './employee-form.component.html',
 styleUrls: ['./employee-form.component.css']})
export class EmployeeFormComponent implements OnInit {
title="Template Driven Form  Validation";
employee={firstName:"Pr",lastName:"chinchole"};
ngOnInit(){  }
}
}
```

Angular Template Driven Form Example

- **app.module.ts**

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { EmployeeFormComponent } from './employee-
form.component.ts';
@NgModule({
 declarations: [    AppComponent, EmployeeFormComponent ],
 imports: [    BrowserModule,    FormsModule ],
bootstrap: [AppComponent]})
export class AppModule { }
```

Angular Template Driven Form Example

- employee-form.component.html

Angular Reactive Form Example

- reactive.component.ts

Angular Reactive Form Example

- reactive.component.html

Angular Reactive Form Example

- **app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule,ReactiveFormsModule  } from '@angular/forms';
import { AppComponent } from './app.component';
import { EmployeeFormComponent } from './employee-form/employee-form.component';
import { ReactiveComponent } from './reactive/reactive.component';
@NgModule({
  declarations: [AppComponent, ReactiveComponent, //components,pipes,directives],
  imports: [ BrowserModule, FormsModule,ReactiveFormsModule //modules],
  providers: [ ], //services
  bootstrap: [AppComponent] //Components
})export class AppModule { }
```

Angular 6 Features

Angular6 Features

- Angular has come out with some amazing new features in version 6.0.0, especially in Angular-cli. Now, with Angular 6, you can easily update your old packages, create native web elements using Angular Elements, and many other things. Let's take a look!
- ng add is a new command in Angular-cli that helps you install and download new packages in your angular apps. It works the same as npm, but it doesn't replace it.

ng add

Ex. ng add @agnular/elements

Angular6 Features

- Declaring the providers inside the service itself
- Before this update, you had to the declare the providers array in app.module.ts

- Now with Angular 6, you can provide your service inside the supervisor itself by putting the providedIn:root property within the "@injectable" decorator.

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root',
})
export class HeroService {
}
```

Angular6 Features

- keyvalue Transforms Object or Map into an array of key value pairs.

```
object: {[key: number]: string} = {2: 'foo', 1: 'bar'};
```

```
<div *ngFor="let item of object | keyvalue">
{{item.key}}:{{item.value}}
</div>
```

Thank You

Atos | Syntel