Angular - Part 2

Presented by
VAISHALI TAPASWI

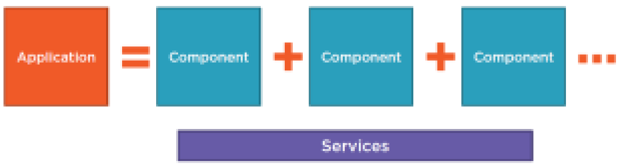
FANDS INFONET Pvt.Ltd.
www.fandsindia.com

Ground Rules

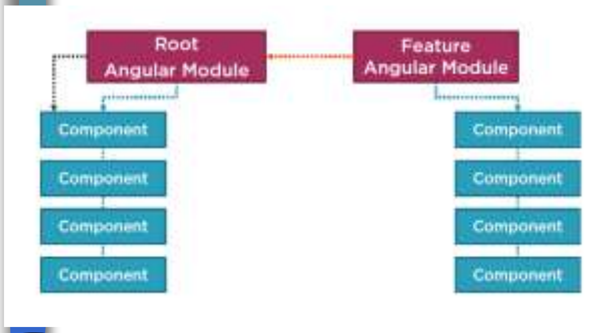
- Turn off cell phone. If you cannot please keep it on silent mode. You can go out and attend your call.
- If you have any questions or issues please let me know immediately.
- Let us be punctual.

Agenda

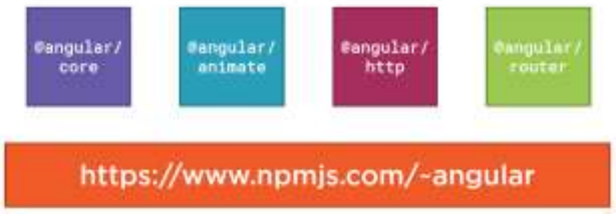
Angular 2 Application



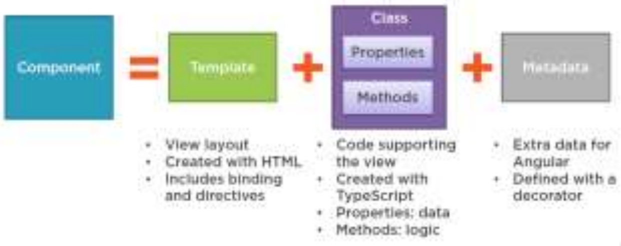
Angular 2 Modules



Angular Is Modular



What Is a Component?

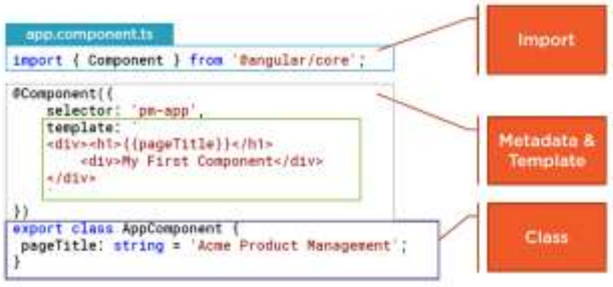


- View layout
- Created with HTML
- Includes binding and directives

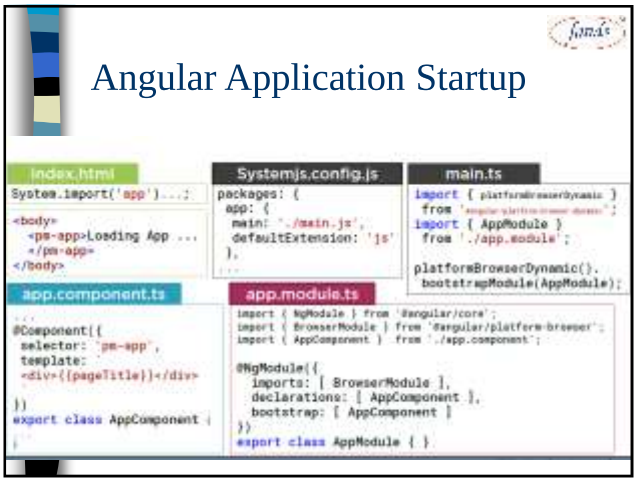
- Code supporting the view
- Created with TypeScript
- Properties: data
- Methods: logic

- Extra data for Angular
- Defined with a decorator

Component



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

@Component({
    selector: 'pm-app',
    template: `
    <div><h1>{{pageTitle}}</h1>
        <div>My First Component</div>
    </div>
    `
})
export class AppComponent {
  pageTitle: string = 'Acme Product Management';
}
```



Angular Application Startup

index.html

```
System.import('app')...;

<body>
  <pm-app>Loading App ...
  </pm-app>
</body>
```

Systemjs.config.js

```
packages: {
 app: {
  main: './main.js',
  defaultExtension: 'js'
 },
 ...
```

main.ts

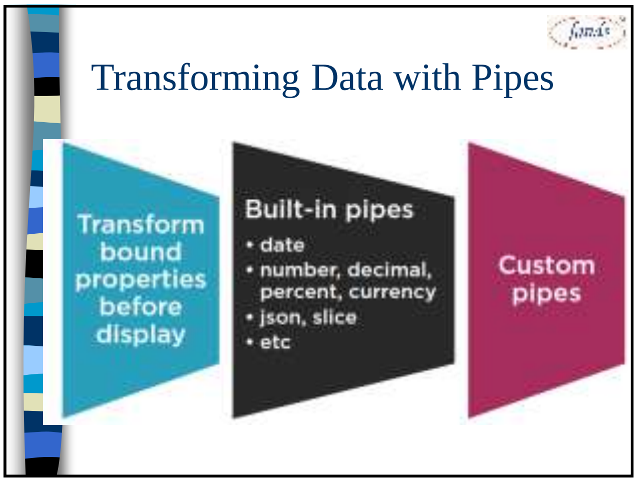
```
import { platformBrowserDynamic }
 from '@angular/platform-browser-dynamic';
import { AppModule }
 from './app.module';

platformBrowserDynamic().
 bootstrapModule(AppModule);
```

app.component.ts

```
...
@Component({
 selector: 'pm-app',
 template: `
 <div>{{pageTitle}}</div>
 `
})
export class AppComponent {
 ...
}
```

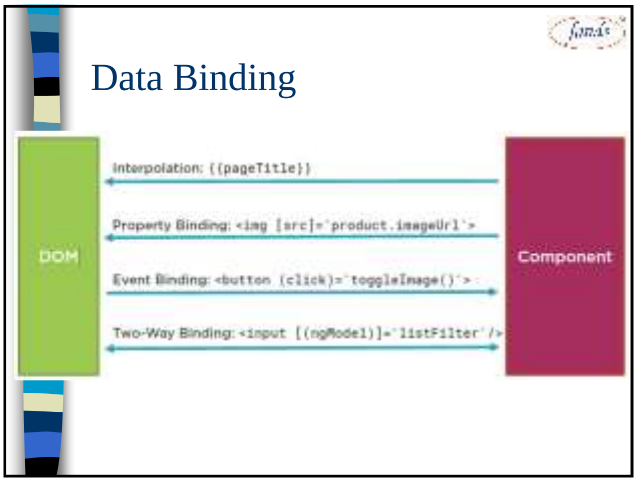
app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';

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



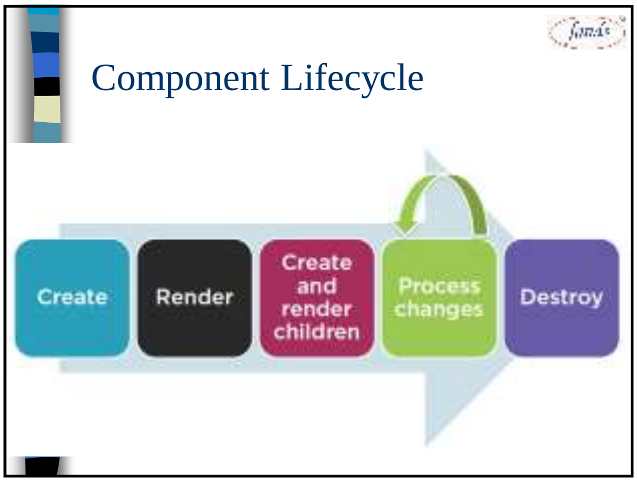
Transforming Data with Pipes

Transform bound properties before display
Built-in pipes
• date
• number, decimal, percent, currency
• json, slice
• etc
Custom pipes



Data Binding

DOM
Component
Interpolation: {{pageTitle}}
Property Binding: <img [src]='product.imageUrl'>
Event Binding: <button (click)='toggleImage()'>
Two-Way Binding: <input [(ngModel)]='listFilter'/>



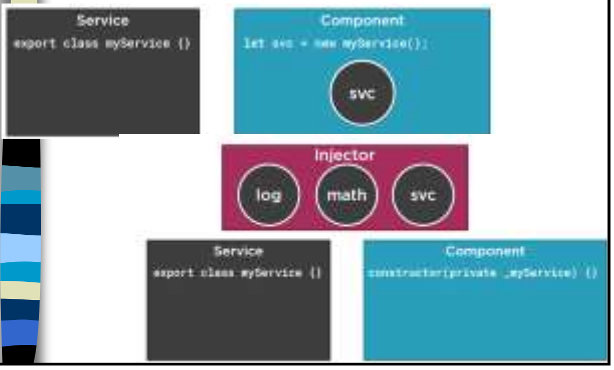
Component Lifecycle

Component Lifecycle Hooks

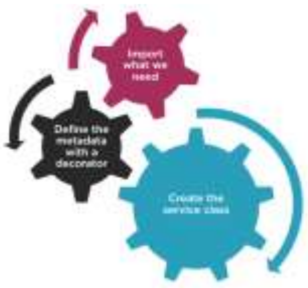
- OnInit
  - Perform component initialization, retrieve data
- OnChanges
  - Perform action after change to input properties
- OnDestroy
  - Perform cleanup

Services and Dependency Injection

How does it work?


Service
export class myService {}
Component
let svc = new myService();
svc
Injector
log
math
svc
Service
export class myService {}
Component
constructor(private _myService) {}

Building a Service



Setting up Angular CLI

- npm install -g @angular/cli:1.6.5
- ng options (Important)
  - – ng build <options...>
    - Builds your app and places it into the output path (dist/ by default).
  - – ng completion <options...>
    - Adds autocomplete functionality to `ng` commands and subcommands.
  - – ng doc
    - Opens the official Angular API documentation for a given keyword.

- npm install -g @angular/cli@1.7.4
- npm install -g @angular/cli

Setting up Angular CLI

- ng options (Important)
  - – ng e2e
    - Run e2e tests in existing project.
  - – ng eject <options...>
    - Ejects your app and output the proper webpack configuration and scripts.
  - – ng generate <schematic> <options...>
  - – ng get <options...>
  - – ng lint <options...>
    - Lints code in existing project.
  - – ng new <options...>
    - Creates a new directory and a new Angular app eg. "ng new [name]“
  - – ng serve <options...>
    - Builds and serves your app, rebuilding on file changes.
  - – ng version

ng generate

- class
- component
- directive
- enum
- guard
- interface
- module
- pipe
- service
- application
- library
- universal

Modules

Creating Modules

- Ng generate module modulename
- Ng generate component modulename/compname

Feature Modules

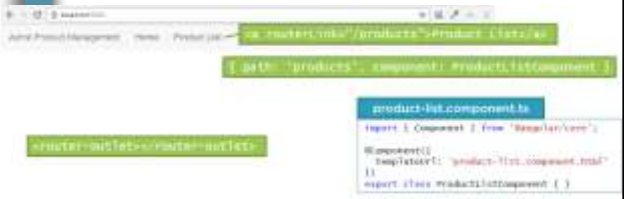
- There are five general categories of feature modules which tend to fall into the following groups:
  - Domain feature modules.
  - Routed feature modules.
  - Routing modules.
  - Service feature modules.
  - Widget feature modules.

Navigation & Routing Basics

How Routing Works?

- Configure a route for each component
- Define options/actions
- Tie a route to each option/action
- Activate the route based on user action
- Activating a route displays the component's view

How Routing Works?



```
<a routerLink="/products">Product List</a>
```

```
{ path: 'products', component: ProductListComponent }
```

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

product-list.component.ts

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

@Component({
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent { }
```

Configuring Routes

app.module.ts

```
...
import { RouterModule } from '@angular/router';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    HttpModule,
    RouterModule.forRoot([], { useHash: true })
  ],
  declarations: [
    ...
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

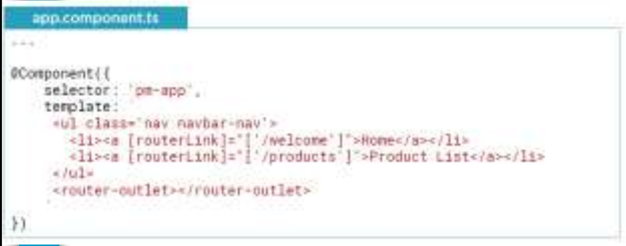
Configuring Routes

```
{ path: 'products', component: ProductListComponent },
{ path: 'product/:id', component: ProductDetailComponent },
{ path: 'welcome', component: WelcomeComponent },
{ path: '', redirectTo: 'welcome', pathMatch: 'full' },
{ path: '**', component: PageNotFoundComponent }
```

Navigating the Application Routes

- Menu option, link, image or button that activates a route
- Typing the Url in the address bar / bookmark
- The browser's forward or back buttons

Tying Routes to Actions and Placing the Views

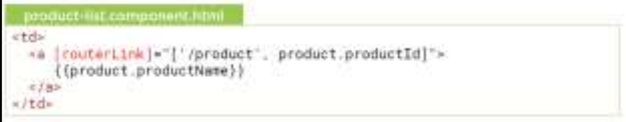

app.component.ts

```
...

@Component({
    selector: 'pm-app',
    template:
     <ul class='nav navbar-nav'>
       <li><a [routerLink]="['/welcome']">Home</a></li>
       <li><a [routerLink]="['/products']">Product List</a></li>
     </ul>
     <router-outlet></router-outlet>

})
```

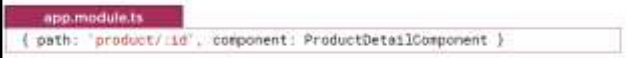
Displaying Components

- Nest-able components
  - Define a selector
  - Nest in another component
  - No route
- Routed components
  - No selector
  - Configure routes
  - Tie routes to actions

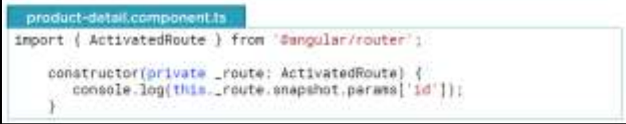
Passing Parameters to a Route


product-list.component.html

```
<td>
  <a [routerLink]="['/product', product.productId]">
     {{product.productName}}
  </a>
</td>
```

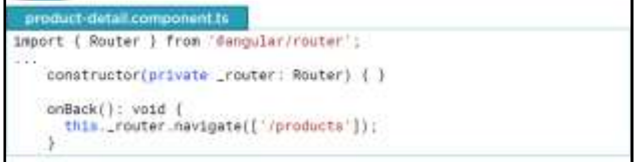

app.module.ts

```
{ path: 'product/:id', component: ProductDetailComponent }
```


product-detail.component.ts

```
import { ActivatedRoute } from '@angular/router';

    constructor(private _route: ActivatedRoute) {
       console.log(this._route.snapshot.params['id']);
    }
```

Activating a Route with Code



product-detail.component.ts

```
import { Router } from '@angular/router';
...
    constructor(private _router: Router) { }

    onBack(): void {
      this._router.navigate(['/products']);
    }
```

Protecting Routes with Guards

- CanActivate
  - Guard navigation to a route
- CanDeactivate
  - Guard navigation from a route
- Resolve
  - Pre-fetch data before activating a route
- CanLoad
  - Prevent asynchronous routing

Modules

- Just using Component from other module (eager loading)
- Loading components from other module using child routes
  - Lazy loading

Forms and Validations

Forms

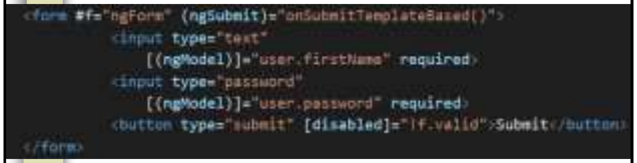
- A form creates a cohesive, effective, and compelling data entry experience. An Angular form coordinates a set of data-bound user controls, tracks changes, validates input, and presents errors.

Form Types

- Template-Driven forms
  - Build forms by writing templates in the Angular template syntax with the form-specific directives and techniques described in this page.
- Model-Driven / Reactive Forms
  - Use the underlying APIs to do them for us. In a sense, instead of binding Object models to directives like template-driven forms, we in fact boot up our own instances inside a component class and construct our own JavaScript models.

Enabling Template Driven Forms

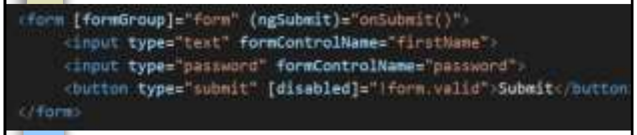
- import {FormsModule} from "@angular/forms";



```
<form #f="ngForm" (ngSubmit)="onSubmitTemplateBased()">
        <input type="text"
            [(ngModel)]="user.firstName" required>
        <input type="password"
            [(ngModel)]="user.password" required>
        <button type="submit" [disabled]="!f.valid">Submit</button>
</form>
```

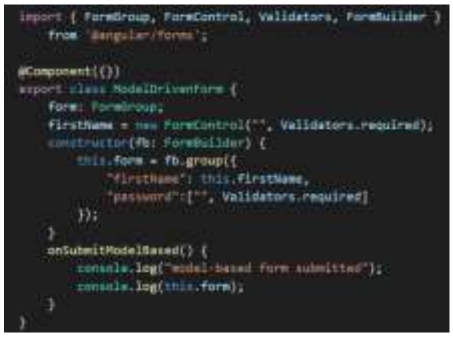
Model Driven Or Reactive Forms

- import {ReactiveFormsModule} from "@angular/forms";



```
<form [formGroup]="form" (ngSubmit)="onSubmit()">
    <input type="text" formControlName="firstName">
    <input type="password" formControlName="password">
    <button type="submit" [disabled]="!form.valid">Submit</button
</form>
```

Controller



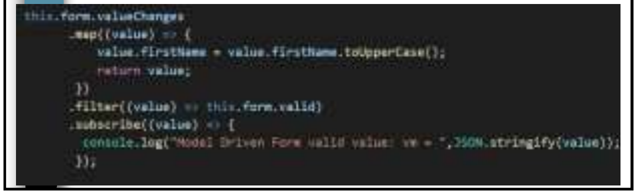
```
import { FormGroup, FormControl, Validators, FormBuilder }
    from '@angular/forms';

@Component({})
export class ModelDrivenForm {
    form: FormGroup;
    firstName = new FormControl("", Validators.required);
    constructor(fb: FormBuilder) {
        this.form = fb.group({
            "firstName": this.firstName,
            "password":["", Validators.required]
        });
    }
    onSubmitModelBased() {
        console.log("model-based form submitted");
        console.log(this.form);
    }
}
```

Functional Reactive Programming

- Form controls and the form itself now provide an Observable-based API. You can think of observables simply as a collection of values over time.
- Both the controls and the whole form itself can be viewed as a continuous stream of values, that can be subscribed to and processed using commonly used functional primitives.

Example

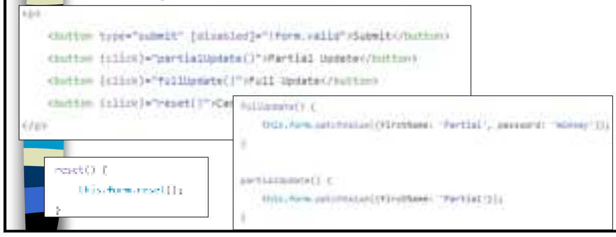
- Convert the first name to uppercase using map and take only the valid form values using filter. This creates a new stream of valid-only values to which we subscribe, by providing a callback that defines how the UI should react to a new valid value.



```
this.form.valueChanges
        .map((value) => {
            value.firstName = value.firstName.toUpperCase();
            return value;
        })
        .filter((value) => this.form.valid)
        .subscribe((value) => {
         console.log("Model Driven Form valid value: vm = ",JSON.stringify(value));
        });
```

Updating Form Values

- We now have APIs available for either updating the whole form, or just a couple of fields.



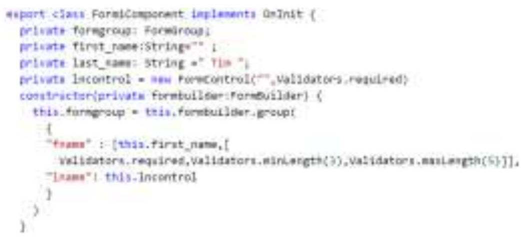
```
<p>
    <button type="submit" [disabled]="!form.valid">Submit</button>
    <button (click)="partialUpdate()">Partial Update</button>
    <button (click)="fullUpdate()">Full Update</button>
    <button (click)="reset()">Ca
</p>
```

```
fullUpdate() {
    this.form.setValue({firstName: 'Partial', password: 'monkey'});
}

partialUpdate() {
    this.form.patchValue({firstName: 'Partial'});
}
```

```
reset() {
    this.form.reset();
}
```

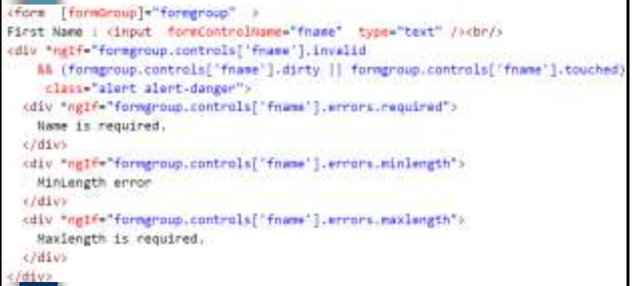
Validations

Reactive Form Validations



```
export class Form1Component implements OnInit {
  private formgroup: FormGroup;
  private first_name:String="" ;
  private last_name: String =" Tim ";
  private lncontrol = new FormControl("",Validators.required)
  constructor(private formbuilder:FormBuilder) {
   this.formgroup = this.formbuilder.group(
     {
     "fname" : [this.first_name,[
      Validators.required,Validators.minLength(3),Validators.maxLength(5)]],
     "lname": this.lncontrol
     }
   )
  }
```

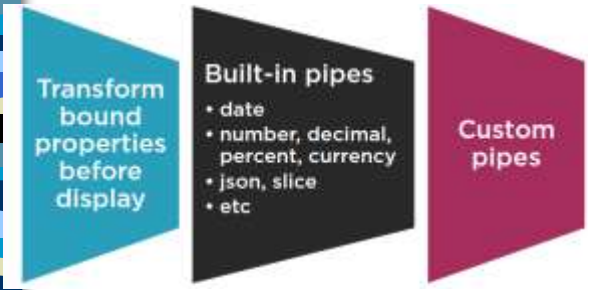
Reactive Form Validations



```
<form  [formGroup]="formgroup"  >
First Name : <input  formControlName="fname"  type="text" /><br/>
<div *ngIf="formgroup.controls['fname'].invalid
    && (formgroup.controls['fname'].dirty || formgroup.controls['fname'].touched)
     class="alert alert-danger">
  <div *ngIf="formgroup.controls['fname'].errors.required">
    Name is required.
  </div>
  <div *ngIf="formgroup.controls['fname'].errors.minlength">
    MinLength error
  </div>
  <div *ngIf="formgroup.controls['fname'].errors.maxlength">
    Maxlength is required.
  </div>
</div>
```

Pipes

Transforming Data with Pipes


Transform bound properties before display
Built-in pipes
• date
• number, decimal, percent, currency
• json, slice
• etc
Custom pipes

Pipes

- A pipe takes in data as input and transforms it to a desired output. In this page, you'll use pipes to transform a component's birthday property into a human-friendly date.
  - Built-in pipes
    - Angular comes with a stock of pipes such as DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, and PercentPipe. They are all available for use in any template.

Pipes

- Parameterizing a pipe
  - A pipe can accept any number of optional parameters to fine-tune its output. To add parameters to a pipe, follow the pipe name with a colon (:) and then the parameter value (such as currency:'EUR'). If the pipe accepts multiple parameters, separate the values with colons (such as slice:1:5)

Pipes

- A pipe is a class decorated with pipe metadata.
- The pipe class implements the PipeTransform interface's transform method that accepts an input value followed by optional parameters and returns the transformed value.
- There will be one additional argument to the transform method for each parameter passed to the pipe. Your pipe has one such parameter: the exponent.
- To tell Angular that this is a pipe, you apply the @Pipe decorator, which you import from the core Angular library.
- The @Pipe decorator allows you to define the pipe name that you'll use within template expressions. It must be a valid JavaScript identifier. Your pipe's name is exponentialStrength

Building a Custom Pipe

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name:'productFilter'
})
export class ProductFilterPipe
                  implements PipeTransform {

  transform(value: IProduct[],
            filterBy: string): IProduct[]{
  }
}
```

Using a Custom Pipe

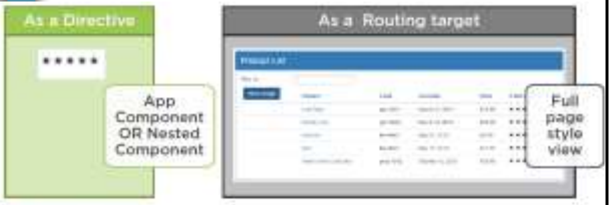
Template

```
<tr *ngFor ='let product of products | productFilter: listFilter'>
```

Module

```
@NgModule({
  imports: [
      BrowserModule,
      FormsModule ],
  declarations: [
      AppComponent,
      ProductListComponent,
      ProductFilterPipe ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

Building Nested Components

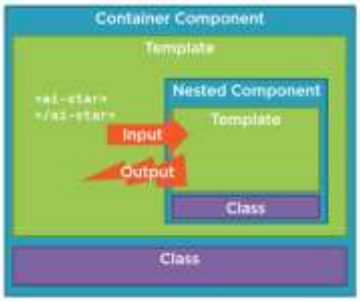


```
<body>
  <mh-app>Loading App ...</mh-app>
</body>
```

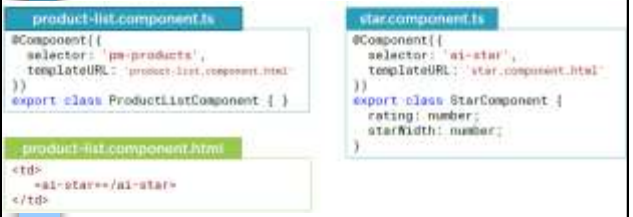
What Makes a Component Nest-able?

- Its template only manages a fragment of a larger view
- It has a selector
- It optionally communicates with its container

Building a Nested Component



Using a Nested Component as Directive


product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: 'product-list.component.html'
})
export class ProductListComponent { }
```

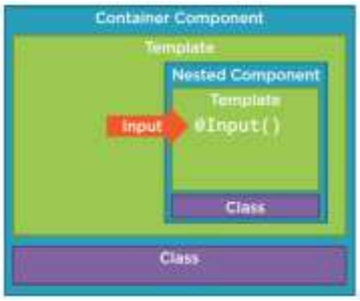
product-list.component.html

```
<td>
   <ai-star></ai-star>
</td>
```

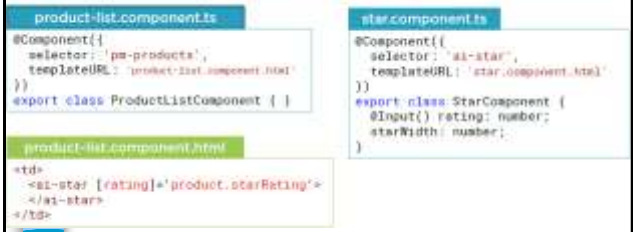
star.component.ts

```
@Component({
  selector: 'ai-star',
  templateURL: 'star.component.html'
})
export class StarComponent {
  rating: number;
  starWidth: number;
}
```

Passing Data to a Nested Component (@Input)



Passing Data to a Nested Component (@Input)


product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: 'product-list.component.html'
})
export class ProductListComponent { }
```

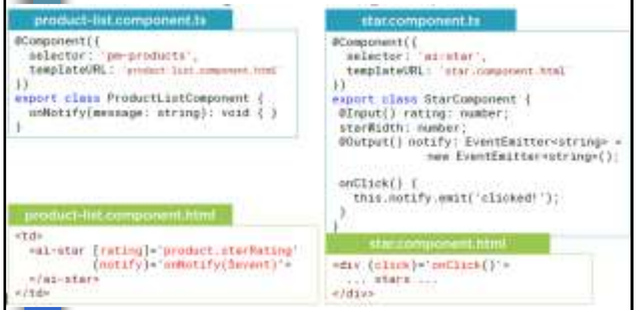
product-list.component.html

```
<td>
  <ai-star [rating]='product.starRating'>
  </ai-star>
</td>
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateURL: 'star.component.html'
})
export class StarComponent {
  @Input() rating: number;
  starWidth: number;
}
```

Raising an Event


product-list.component.ts

```
@Component({
  selector: 'pm-products',
  templateURL: 'product-list.component.html'
})
export class ProductListComponent {
  onNotify(message: string): void { }
}
```

product-list.component.html

```
<td>
  <ai-star [rating]='product.starRating'
           (notify)='onNotify($event)'>
  </ai-star>
</td>
```

star.component.ts

```
@Component({
  selector: 'ai-star',
  templateURL: 'star.component.html'
})
export class StarComponent {
 @Input() rating: number;
 starWidth: number;
 @Output() notify: EventEmitter<string> =
              new EventEmitter<string>();

 onClick() {
   this.notify.emit('clicked!');
 }
}
```

star.component.html

```
<div (click)='onClick()'>
  ... stars ...
</div>
```

Component Styling

Component Styles

- Angular applications are styled with standard CSS. That means you can apply everything you know about CSS stylesheets, selectors, rules, and media queries directly to Angular applications.
- Additionally, Angular can bundle *component styles* with components, enabling a more modular design than regular stylesheets.

Using component styles

- @Compoent
  - styles: ['h1 { font-weight: normal; }']
  - stylesUrl :['./../t1.css']
- The styles specified in @Component metadata apply only within the template of that component.
- They are not inherited by any components nested within the template nor by any content projected into the component.

Loading Component Styles

- By setting styles or styleUrls metadata.
- Inline in the template HTML.
  - You can specify more than one styles file or even a combination of styles and styleUrls.
- With CSS imports.

External and global style files

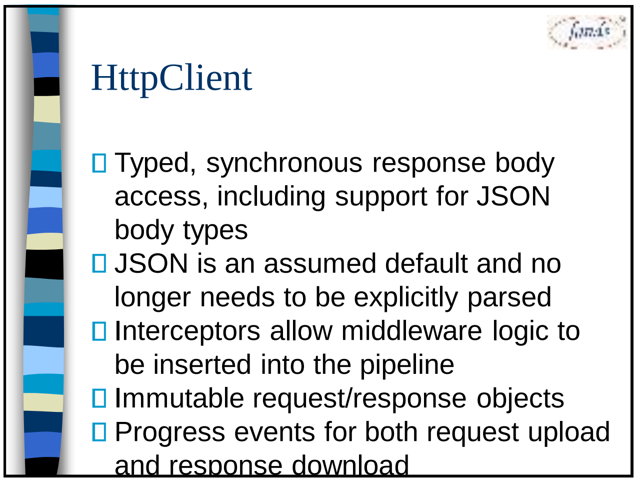
- When building with the CLI, you must configure the angular.json to include all external assets, including external style files.
- Register global style files in the styles section which, by default, is pre-configured with the global styles.css file.

Non-CSS style files

- If you're building with the CLI, you can write style files in sass, less, or stylus and specify those files in the @Component.styleUrls metadata with the appropriate extensions (.scss, .less, .styl) as in the following example:

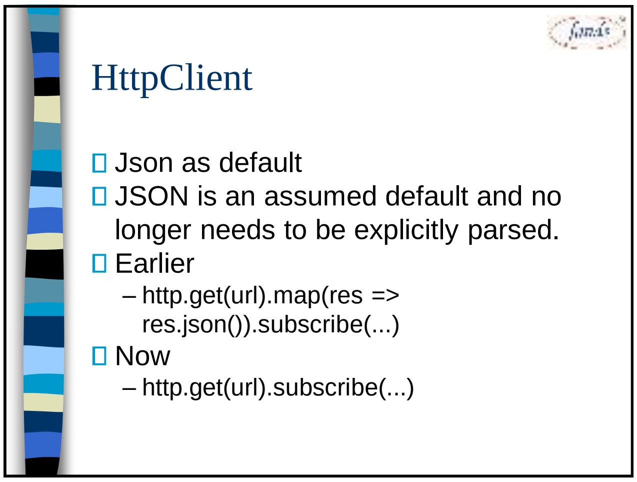
```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
```

HttpClient



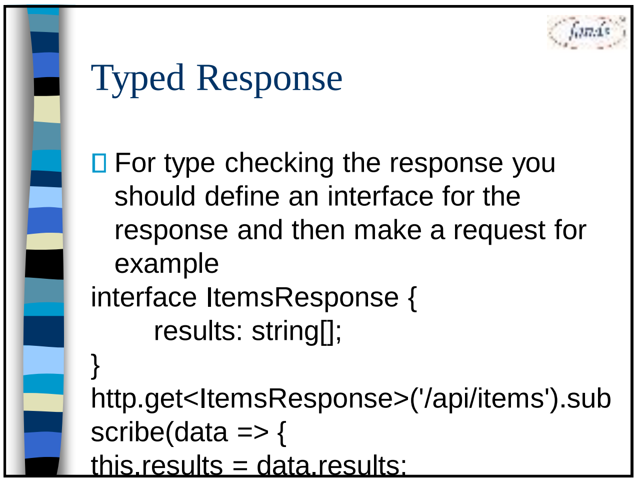
HttpClient

- Typed, synchronous response body access, including support for JSON body types
- JSON is an assumed default and no longer needs to be explicitly parsed
- Interceptors allow middleware logic to be inserted into the pipeline
- Immutable request/response objects
- Progress events for both request upload and response download



HttpClient

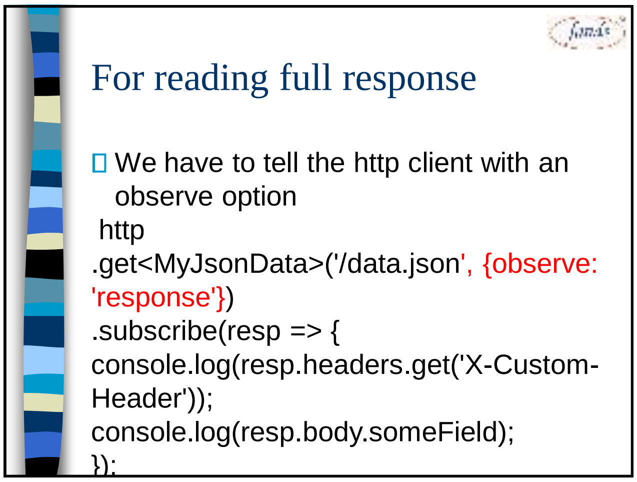
- Json as default
- JSON is an assumed default and no longer needs to be explicitly parsed.
- Earlier
  - http.get(url).map(res => res.json()).subscribe(...)
- Now
  - http.get(url).subscribe(...)



Typed Response

- For type checking the response you should define an interface for the response and then make a request for example

```
interface ItemsResponse {
        results: string[];
}
http.get<ItemsResponse>('/api/items').sub
scribe(data => {
this.results = data.results;
```



For reading full response

- We have to tell the http client with an observe option

```
 http
.get<MyJsonData>('/data.json', {observe:
'response'})
.subscribe(resp => {
console.log(resp.headers.get('X-Custom-
Header'));
console.log(resp.body.someField);
});
```

Non JSON Data

- We have to specify the response type

```
http
.get('/textfile.txt', {responseType: 'text'})
.subscribe(data => console.log(data));
```

Headers and Params

- Setting up the headers

```
http.post('/api/items/add', body, {
headers: new HttpHeaders().set('Authorization',
'my-auth-token'),
}) .subscribe();
```

- Setting up the params

```
http.post('/api/items/add', body, params: new
HttpParams().set('id', '3'),}).subscribe();
```

Interceptor Support

- A major feature of @angular/common/http is interception, the ability to declare interceptors which sit in between your application and the backend.
- When your application makes a request, interceptors transform it before sending it to the server, and the interceptors can transform the response on its way back before your application sees it. This is useful for everything from authentication to logging.

Compilation

Compilation Options

- Angular offers two ways to compile your application:
  - *Just-in-Time* (JIT), which compiles your app in the browser at runtime
    - Default
    - ng build
    - ng serve
  - *Ahead-of-Time* (AOT), which compiles your app at build time
    - ng build –aot
    - ng serve --aot

JiT

- Flow of events with Just-in-Time Compilation
  - Development of Angular application with TypeScript.
  - Compilation of the application with tsc.
  - Bundling.
  - Minification.
  - Deployment.

JiT

- Once we've deployed the app and the user opens browser, will go through the following steps (without strict CSP):
  - Download all the JavaScript assets.
  - Angular bootstraps.
  - Angular goes through the JiT compilation process, i.e. generation of JavaScript for each component in our application.
  - The application gets rendered.

AoT

- Flow of events with Ahead-of-Time Compilation
  - Development of Angular application with TypeScript.
  - Compilation of the application with ngc.
  - Performs compilation of the templates with the Angular compiler and generates (usually) TypeScript.
  - Compilation of the TypeScript code to JavaScript.
  - Bundling.
  - Minification.
  - Deployment.

AoT

- Although the above process seems lightly more complicated the user goes only through the steps:
  - Download all the assets.
  - Angular bootstraps.
  - The application gets rendered.

Why compile with AOT?

- Faster rendering
- Fewer asynchronous requests
- Smaller Angular framework download size
- Detect template errors earlier
- Better security

Deployment

NPM

- npm opens up an entire world of JavaScript talent for you and your team. It's the world's largest software registry
- The registry contains over 600,000 packages (building blocks of code).
- Can use npm to share and borrow packages and many organizations use npm to manage private development as well.

NPM

- http://www.npmjs.com
  OR
- Create a User Account / Login
  – Npm adduser/ npm login
- Check User Account / Logout
  – Npm whoami, Npm logout
- https://www.npmjs.com/~yourusername

Testing

Testing

- Controller
- Filter
- Pipe
- Directive
- Service

End to End (E2E)Testing

- As applications grow in size and complexity, it becomes unrealistic to rely on manual testing to verify the correctness of new features, catch bugs and notice regressions. Unit tests are the first line of defense for catching bugs, but sometimes issues come up with integration between components which can't be captured in a unit test. End-to-end tests are made to find these problems.
- Note: In the past, end-to-end testing could be done with a deprecated tool called Angular Scenario Runner. That tool is now in maintenance mode.

E2E Testing

- Work with Protractor, an end to end test runner which simulates user interactions that will help you verify the health of your Angular application.
- Protractor is a Node.js program, and runs end-to-end tests that are also written in JavaScript and run with node. Protractor uses WebDriver to control browsers and simulate user actions.
- Protractor uses Jasmine for its test syntax.

Testing - Practicals

- Additional tools for testing Angular applications
  - Jasmine
  - Karma
  - Angular-mocks

Jasmine

- Jasmine is a behavior driven development framework for JavaScript that has become the most popular choice for testing Angular applications. Jasmine provides functions to help with structuring your tests and also making assertions. As your tests grow, keeping them well structured and documented is vital, and Jasmine helps achieve this.

Jasmine

- In Jasmine we use the describe function to group our tests together:

```
describe("sorting the list of users", function() {   //
   individual tests go here });
```

- And then each individual test is defined within a call to the it function:

```
describe('sorting the list of users', function() {
  it('sorts in descending order by default', function() {
    // your test assertion goes here
  });
});
```

Jasmine

- Finally, Jasmine provides matchers which let you make assertions:

```
describe('sorting the list of users', function() {
  it('sorts in descending order by default', function() {
    var users = ['jack', 'igor', 'jeff'];
    var sorted = sortUsers(users);
    expect(sorted).toEqual(['jeff', 'jack', 'igor']);
  });
});
```

Karma

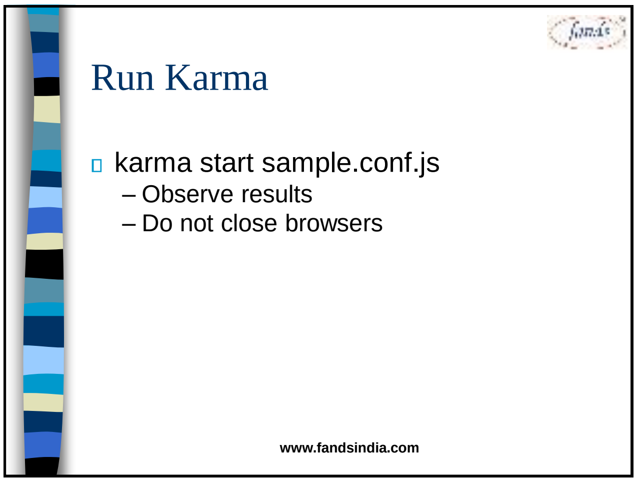
- Karma is a JavaScript command line tool that can be used to spawn a web server which loads your application's source code and executes your tests. You can configure Karma to run against a number of browsers, which is useful for being confident that your application works on all browsers you need to support. Karma is executed on the command line and will display the results of your tests on the command line once they have run in the browser.
- Karma is a NodeJS application, and should be installed through npm. Full installation instructions are available on the Karma website.

Karma

- “karma init sample.conf.js” command will create sample.conf.js after asking preferences
  - which test framework to use (Jasmine in our case),
  - which files to monitor and use (this includes at least)
    - angular.js
    - angular-mocks.js (contains the definition for **module** and **inject**)
    - our own code to test
    - the files containing the tests or specifications
  - what browsers to use to run the test against (we’ll stick with Chrome)
  - The port on which the server is listening
  - whether to run the test automatically or manually when any of the monitored files changes

Sample File

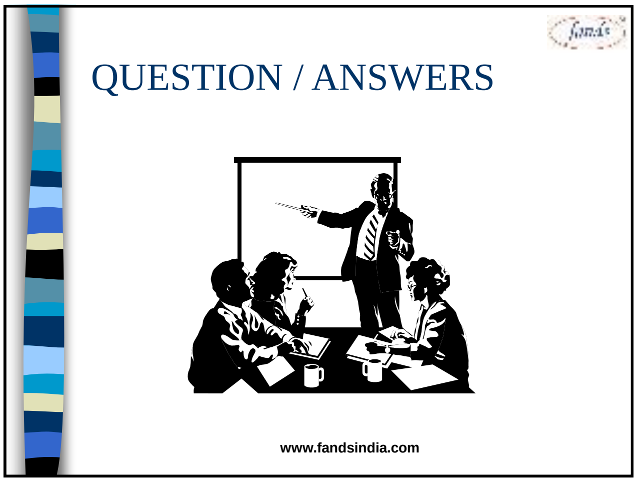
```
module.exports = function(config) {
  config.set({
    basePath: 'test',
    frameworks: ['jasmine'],
    files: [  'src/hw*.js',           'spec/*Spec.js'     ],
    exclude: [    ],
    preprocessors: {    },
    reporters: ['progress'],
    port: 9876,
    colors: true,
    config.LOG_DEBUG
    logLevel: config.LOG_INFO,
    autoWatch: true,
    browsers: ['Chrome', 'Firefox', 'IE'],
      singleRun: true
  });
};
```



Run Karma

- karma start sample.conf.js
  - Observe results
  - Do not close browsers

www.fandsindia.com



QUESTION / ANSWERS

www.fandsindia.com

THANKING YOU !

www.fandsindia.com