

Angular

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.

www.fandsindia.com

Agenda

Discuss

Rich Internet Applications
AJAX

www.fandsindia.com

www.fandsindia.com



TypeScript



Typescript

- Why
 - Javascript is dynamic type
 - Pro- can hold any object, type on the fly
 - Con- Can get messy over the time
 - Migration from server-side to client-side will be hard
 - Hard to manage, difficult to ensure property types
- What
 - Any valid JavaScript is a typescript
 - Typescriptlang.org
 - Typescript lets you write JavaScript the way you really want to.
 - Typescript is a typed superset of JavaScript that compiles to plain JavaScript.
 - Any browser. Any host. Any OS. Open Source.



Typescript Alternatives

- Pure JavaScript
- Apply JavaScript patterns
 - Functions as abstractions
 - Functions to build modules
 - Functions to avoid global variables
- CoffeeScript
- Dart



Typescript Key Features

- Supports standard JavaScript code
- Provide static typing
- Encapsulation through classes and modules
- Support for constructors, properties, functions
- Define interfaces
- Lambda style function support
- Intellisense and syntax checking

Typescript tools

- Typescript playground
- Visual Studio
- sublime
- Node.js
- WebStorm
- Eclipse
- Vi
- IntelliJ
- Emacs

Typescript to JavaScript

```

// TypeScript
class Greeter {
  greeting: string;
  constructor(message: string) {
    this.greeting = message;
  }
  greet() {
    return `Hello, ${this.greeting}`;
  }
}

var greeter = new Greeter("world");

var button = document.getElementById("button");
button.addEventListener("click", function() {
  let btnClick = function() {
    alert(greeter.greet());
  };
  document.body.appendChild(btnClick);
});

// JavaScript
var Greeter = (function () {
  function Greeter(message) {
    this.greeting = message;
  }
  Greeter.prototype.greet = function () {
    return `Hello, ${this.greeting}`;
  };
  return Greeter;
})();

var greeter = new Greeter("world");

var button = document.getElementById("button");
button.addEventListener("click", function () {
  let btnClick = function () {
    alert(greeter.greet());
  };
  document.body.appendChild(btnClick);
});
    
```

Typescript BasicTypes

- Boolean
- Number
- String
- Array
- Enum
- Enum as Bit Flag 1,2,4,8,16,32,64,128 and so on
- Any
- Void

Typescript Annotation

- Type Annotation
 - var [identifier]:[type annotation]=value
 - var [identifier]:[type annotation];
 - var [identifier]=value;

```

let isAdult: boolean = false; //boolean
var latitude: number = 41.88; //number
var name: string = "typescript"; //string
enum color { Red, Blue, Green } //enum
var colors: color = color.Red;
enum color2 { Red = 1, Blue = 2, Green = 4 } //enum with flag
var arrNum: number[] = [1, 2, 3]; //array of number
var listNumbers: Array<number> = [1, 2, 3]; //array of number using Array
var anyType: any = 4; //number
anyType = "typescript"; //assigning dynamic type
var listAny: any[] = [1, true, "typescript"]; //dynamic type array
listAny[1] = 100; //changing boolean to number
    
```

Typescript Interfaces

- Interfaces are used at design time to provide auto completion and at compile time to provide type checking
- Supported features
 - Optional properties
 - Function Types
 - Array Types
 - Class Types
 - Extending Interfaces
 - Hybrid Types

Typescript Interfaces

```
// Interface with a single optional property
interface UserRegistration {
  firstName: string;
  lastName: string;
  middleName?: string;
}

// Extending Interface
interface User {
  getAge(): number;
}

// Overloading Interface
interface User {
  getAge(): number;
  getAge(): number;
  getAge(): number;
}

// Function Type
interface User {
  (name: string, password: string): boolean;
}

// Array Type
interface User {
  [index: number]: string;
  length: number;
}

// Hybrid Type
interface User {
  (name: string): string;
  [index: number]: string;
  length: number;
}
```

Typescript Classes

- Object-oriented class based approach
- Key features
 - Inheritance
 - Private/public modifiers
 - Accessors
 - Static properties
 - Constructor functions
 - Using class as an interface

Typescript Classes

```
// Class
class User {
  constructor(public firstName: string, public lastName: string) {}
}

// Inheritance
class Admin extends User {
  constructor(public firstName: string, public lastName: string, public password: string) {
    super(firstName, lastName);
  }
}

// Accessors
class User {
  private _firstName: string;
  private _lastName: string;

  get firstName(): string {
    return this._firstName;
  }

  set firstName(value: string) {
    this._firstName = value;
  }

  get lastName(): string {
    return this._lastName;
  }

  set lastName(value: string) {
    this._lastName = value;
  }
}
```

Typescript Generics

- Supports generic type variables, types, interfaces, classes and constraints

```
function Identity<T>(arg: T): T {
    return arg;
}

var output = Identity<string>('hello') // Identity<'string'>
function loggingIdentity<T>(arg: T): T { // T[] | T[] can be written as Array<T>
    console.log(arg, arg.length); // Array has a .length, so no need for arg
    return arg;
}
```

Typescript Modules

- Encapsulate variables, interfaces, and classes
 - Define unique namespaces
 - Organize symbols and identifiers into a logical namespace hierarchy
 - Similar to namespaces/packages
- Splitting across files
 - Multiple files can use the same module name
- One file can contain multiple module
- Can define Alias to module
- Transpiles to IIFE
- Can define modules as internal or external
- External modules required only when used with node.js and require.js

Typescript Modules

```
module myModule { //module can be to namespace file
    export interface User {
        get姓(): string;
        get姓(): string;
    }
}

module myModule { //namespace module
    export class User {
        constructor() {
            this.姓 = '1';
        }
        get姓(): string {
            return this.姓;
        }
    }
}

module myModule { //namespace module
    export class User {
        constructor() {
            this.姓 = '1';
        }
        get姓(): string {
            return this.姓;
        }
    }
}

module myModule { //namespace module
    export class User {
        constructor() {
            this.姓 = '1';
        }
        get姓(): string {
            return this.姓;
        }
    }
}
```

Typescript Definition Files

- Describes the types defined in external libraries
- .d.ts
- Not deployed
- Usually from DefinitelyTyped
- TypeScript Definition manager (tsd)
 - Specialized package manager
 - Locates and installs typescript definition files(d.ts)
 - From the definitelytyped repository

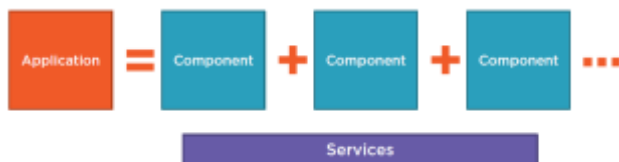
Why Angular?

- Expressive HTML
- Powerful Data Binding
- Modular By Design
- Built-in Back-End Integration

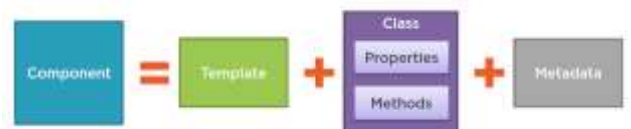
Why Angular 2?

- Built for Speed
- Modern
- Simplified API
- Enhances Productivity

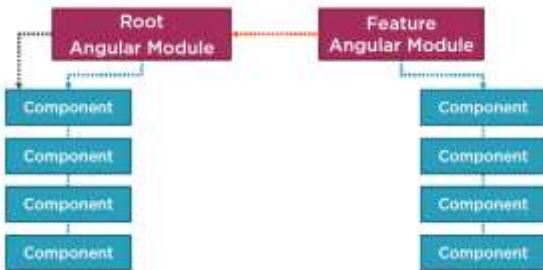
Angular 2 Application



Component



Angular 2 Modules



JavaScript Language Specification

- ECMAScript (ES)
- ES 3
- ES 5
- ES 2015 (formally known as ES 6)
 - Must be transpiled

Selecting a language

- ES 5
 - Runs in the browser
 - No compile required
- ES 2015
 - Lots of new features (classes, let, arrow, etc.)
- TypeScript
 - Superset of JavaScript
 - Strong typing
 - Great IDE tooling
- Dart
 - No JavaScript

What Is TypeScript?

- Open Source Language
- Superset of JavaScript
- Transpiles to plain JavaScript
- Strongly typed
 - TypeScript type definition files (*.d.ts)
- Class-based object-orientation

Setting up environment

- Npm
 - Node Package Manager
 - Command Line Utility
 - Installs libraries, packages and applications
- Set up the Angular 2 application

Setting up an Angular 2 Application

- Create an application folder
- Add package definition and configuration files
- Install the packages
- Create the app's Angular Module
- Create the main.ts file
- Create the host Web page (index.html)

Setting up an Angular 2 Application

- Manually perform each step
 - www.angular.io Quick Start
- Download the results of these steps
 - <https://github.com/angular/quickstart>
- AngularCli
 - <https://github.com/angular/angular-cli>
- Starter files

Modules



TypeScript

Export

```
product.ts  
  
export class Product {  
}
```

Transpile

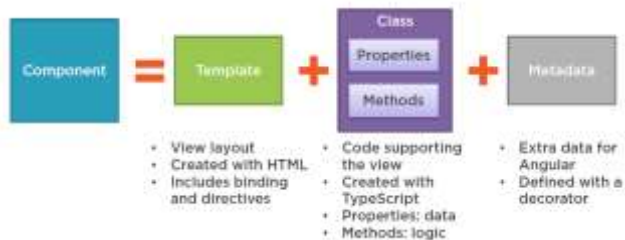
```
product.js  
  
function Product() {  
}
```

Import

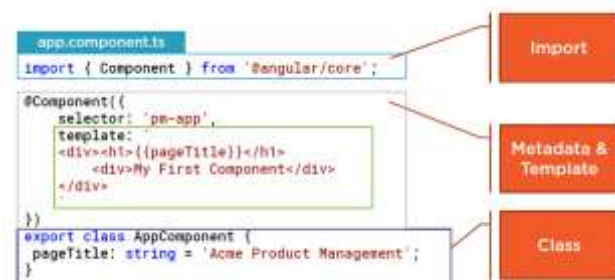
```
product-list.ts  
  
import { Product } from  
  './product'
```

Starting with Components

What Is a Component?



Component



Creating the Component Class

```
app.component.ts
export class AppComponent {
  pageTitle: string = 'Acme Product Management';
}
```

Diagram labels:

- export keyword
- class keyword
- Class Name
- Component Name when used in code

Defining the Metadata

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

Decorator

- A function that adds metadata to a class, its members, or its method arguments.
- Prefixed with an @.
- Angular provides built-in decorators.
- @Component()

Defining the Metadata

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

Diagram labels:

- Component decorator
- Directive Name used in HTML
- View Layout
- Binding

Importing What We Need

- Before we use an external function or class, we define where to find it
- Import statement
- Import allows us to use exported members from external ES modules
- Import from a third-party library, our own ES modules, or from Angular

Angular Is Modular



Importing What We Need

```
app.component.ts
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';
}
```

Diagram labels pointing to the import statement:

- import keyword
- Angular library module name
- Member name

Completed Component

```
app.component.ts
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';
}
```

Bootstrapping Our App Component

- Load the root component (bootstrapping)
- Host the application

Angular Application Startup

index.html	Systemjs.config.js	main.ts
<pre>System.import('app')...; <body> <pm-app>Loading App ... </pm-app> </body></pre>	<pre>packages: { app: { main: './main.js', defaultExtension: '.js' }, ... }</pre>	<pre>import { platformBrowserDynamic } from '@angular/platform-browser-dynamic'; import { AppModule } from './app.module'; platformBrowserDynamic(). bootstrapModule(AppModule);</pre>
app.component.ts	app.module.ts	
<pre>@Component({ selector: 'pm-app', template: ` <div>{{pageTitle}}</div> ` }) export class AppComponent { ... }</pre>	<pre>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 { }</pre>	

Component Recap

- Class -> Code
 - Clear name
 - Use PascalCasing
 - Append "Component" to the name
 - Export keyword
 - Data in properties
 - Appropriate data type
 - Appropriate default value
 - camelCase with first letter lowercase
 - Logic in methods
 - camelCase
 - with first letter lowercase
- Decorator -> Metadata
 - Component decorator
 - Prefix with @; Suffix with ()
 - Selector : Component name in HTML
 - Prefix for clarity
 - Template : View's HTML
 - Correct HTML syntax
- Import what we need
 - Defines where to find the members that this component needs
 - Import keyword
 - Member name
 - Correct spelling/casing
 - Module path
 - Enclose in quotes
 - Correct spelling/casing

Templates and Directives

```
app.component.ts
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';
}
```

Defining a Template



Directives

□ `<selector>content</selector>`



Bindings

□ Coordinates communication between the component's class and its template and often involves passing data.



Interpolation



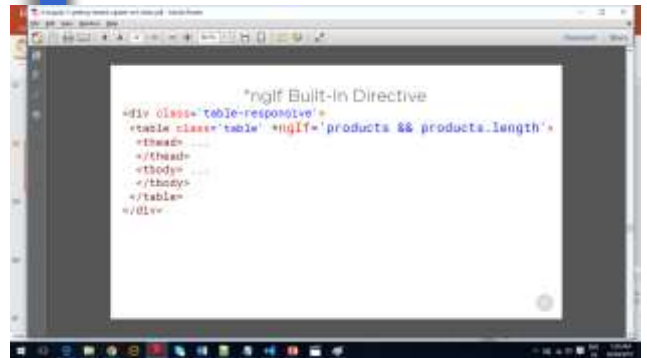
Directives

□ Custom HTML element or attribute used to power up and extend our HTML.

- Built-In
 - *ngIf, *ngFor
- Custom

```
app.component.ts
@Component({
  selector: 'pm-app',
  template:
    <div><h1>{{pageTitle}}</h1>
    <pm-products></pm-products>
</div>
})
export class AppComponent { }
```

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



Built-in Directives

```
<div class='table-responsive'>
  <table class='table' *ngIf='products && products.length'>
    <thead> ...
    </thead>
    <tbody> ...
    </tbody>
  </table>
</div>
```

```
<tr *ngFor='let product of products'>
  <td></td>
  <td>{{ product.productName }}</td>
```

Template
input variable

Data Binding



Binding

- Property Binding
 - ``
 - ``
 - ``
- Event Binding
 - `<button(click)='toggleImage()'>`

Two-way Binding

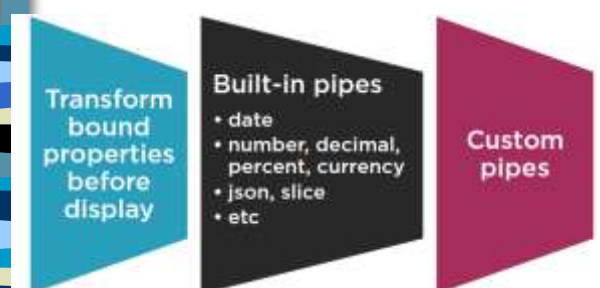
- FormsModule



Data Binding



Transforming Data with Pipes



Pipe Examples

- ❑ {{ product.productCode| lowercase }}
- ❑
- ❑ {{ product.price| currency | lowercase }}
- ❑ {{ product.price| currency:'USD':true:'1.2-2' }}

ngModel

```
product @ui.component.html  
<div class='col-md-4'>  
  <input type='text'  
    [(ngModel)]='listFilter' />  
</div>
```

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

Improving Components

- ❑ Strong typing & interfaces
- ❑ Encapsulating styles
- ❑ Lifecycle hooks
- ❑ Custom pipes
- ❑ Relative Paths with Module Id

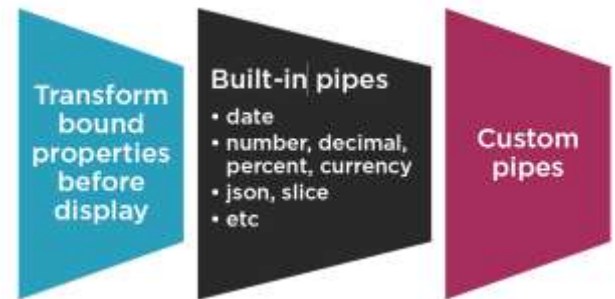
Strong Typing

```
export class ProductListComponent {  
  pageTitle: string = 'Product List';  
  showImage: boolean = false;  
  listFilter: string = 'cart';  
  message: string;  
  
  products: any[] = [...];  
  
  toggleImage(): void {  
    this.showImage = !this.showImage;  
  }  
  
  onRatingClicked(message: string): void {  
    this.message = message;  
  }  
}
```


Interface

- A specification identifying a related set of properties and methods.
- A class commits to supporting the specification by implementing the interface.
- Use the interface as a data type.
- Development time only!
- Interface

Transforming Data with Pipes

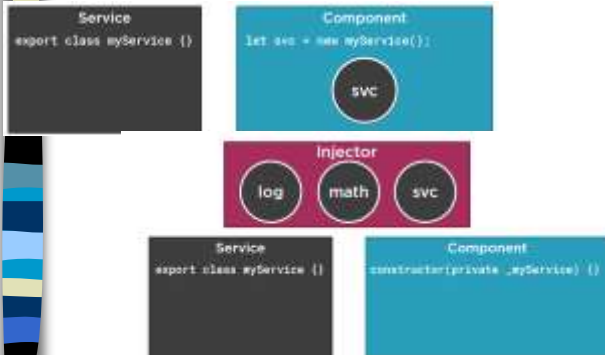


Services and Dependency Injection

Service

- A class with a focused purpose.
- Used for features that:
 - Are independent from any particular component
 - Provide shared data or logic across components
 - Encapsulate external interactions

How does it work?



Dependency Injection

- A coding pattern in which a class receives the instances of objects it needs (called dependencies) from an external source rather than creating them itself.

Building a Service



Building a Service

```
product.service.ts
import { Injectable } from '@angular/core'

@Injectable()
export class ProductService {

  getProducts(): IProduct[] {

  }

}
```

Registering a Provider

```
app.component.ts
...
import { ProductService } from './products/product.service';

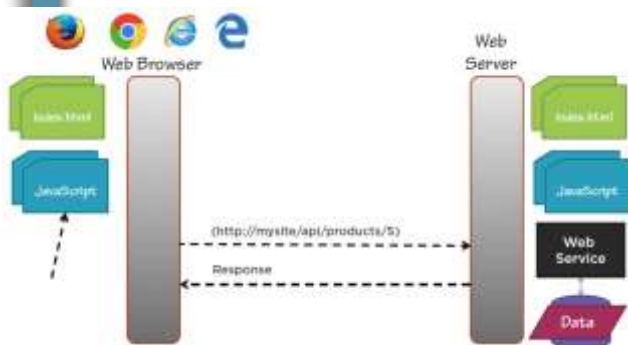
@Component({
  selector: 'pm-app',
  template: `
    <div><h1>{{pageTitle}}</h1>
    <pm-products></pm-products>
  </div>
  `,
  providers: [ProductService]
})
export class AppComponent { }
```

Injecting the Service

```
product-list.component.ts
...
import { ProductService } from './products/product.service';

@Component({
  selector: 'pm-products',
  templateUrl: 'product-list.component.html'
})
export class ProductListComponent {
  private _productService;
  constructor(productService: ProductService) {
    _productService = productService;
  }
}
```

Retrieving Data Using HTTP



Observables and Reactive Extensions

- Help manage asynchronous data
- Treat events as a collection
 - An array whose items arrive asynchronously over time
- Are a proposed feature for ES 2016
- Use Reactive Extensions (RxJS)
- Are used within Angular

Observable Operators

- Methods on observables that compose new observables
- Transform the source observable in some way
- Process each value as it is emitted
- Examples: map, filter, take, merge, ...

Promise vs Observable

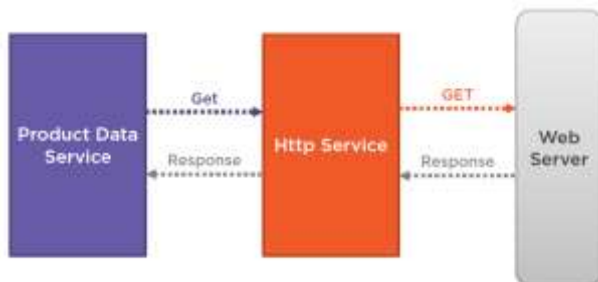
Promise

- Provides a single future value
- Not lazy
- Not cancellable

Observable

- Emits multiple values over time
- Lazy
- Cancellable
- Supports map, filter, reduce and similar operators

Sending an Http Request



Sending an Http Request

```
product.service.ts
...
import { Http } from '@angular/http';

@Injectable()
export class ProductService {
  private _productUrl = 'www.myWebService.com/api/products';

  constructor(private _http: Http) { }

  getProducts() {
    return this._http.get(this._productUrl);
  }
}
```

Registering the Http Service Provider

```
app.module.ts
...
import { HttpClientModule } from '@angular/http';

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

Sending an Http Request

```
product.service.ts
...
import { Http } from '@angular/http';

@Injectable()
export class ProductService {
  private _productUrl = 'www.myWebService.com/api/products';

  constructor(private _http: Http) { }

  getProducts() {
    return this._http.get(this._productUrl);
  }
}
```

Sending an Http Request with observable

```
product.service.ts
...
import { Http, Response } from '@angular/http';
import { Observable } from 'rxjs/Observable';
import 'rxjs/add/operator/map';

@Injectable()
export class ProductService {
  private _productUrl = 'www.myWebService.com/api/products';

  constructor(private _http: Http) { }

  getProducts(): Observable<IProduct[]> {
    return this._http.get(this._productUrl)
      .map((response: Response) => <IProduct[]>response.json());
  }
}
```

Exception Handling

```
product.service.ts
...
import 'rxjs/add/operator/do';
import 'rxjs/add/operator/catch';

...
getProducts(): Observable<IProduct[]> {
  return this._http.get(this._productUrl)
    .map((response: Response) => <IProduct[]>response.json())
    .do(data => console.log('All: ' + JSON.stringify(data)))
    .catch(this.handleError);
}

private handleError(error: Response) {
}
```

Subscribing to an Observable

```
x.then(valueFn, errorFn) //Promise
x.subscribe(valueFn, errorFn) //Observable
x.subscribe(valueFn, errorFn, completeFn) //Observable
let sub = x.subscribe(valueFn, errorFn, completeFn)
```

product-list.component.ts

```
ngOnInit(): void {
  this._productService.getProducts()
    .subscribe(products => this.products = products,
      error => this.errorMessage = <any>error);
}
```

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

```
RouterModule.forRoot([
  { path: '', redirectTo: '/products', pathMatch: 'full' },
  { path: 'products', loadChildren: () => import('./products/product-list.component') },
], { usePathMatching: true })
```

How Routing Works?

```
RouterModule.forRoot([
  { path: 'products', component: ProductListComponent }
], { usePathMatching: true })
```

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,
    HttpClientModule,
    RouterModule.forRoot([], { useHash: true })
  ],
  declarations: [
    ...
  ],
  bootstrap: [ AppComponent ]
})
export class AppModule { }
```

QUESTION / ANSWERS



www.fandsindia.com

THANKING YOU !



www.fandsindia.com