Coding Bootcamp

Angular

TECHNOLOGY

Working with Angular Services and Router

simplilearn

Learning Objectives

By the end of this lesson, you will be able to:

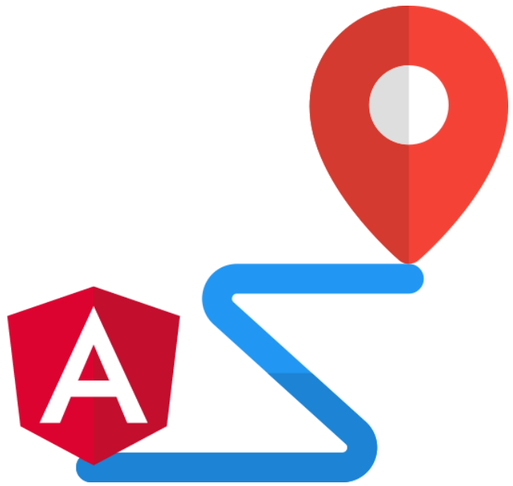
- Implement Angular routing for seamless navigation between web pages
- Demonstrate routing mechanisms between parent and child
- Implement services and injectables to transform data

TECHNOLOGY

Routing in Angular

Angular Router

Routing refers to navigating between pages or views.



Routing redirects the user from one component view to another.

Developers use it in a single-page app (SPA) to show or hide portions of the display depending on the user's navigation.

Angular Router

Routing refers to navigating between pages or views.

It allows developers to build single-page, multi-state, multi-view applications.

It allows client-side navigation.

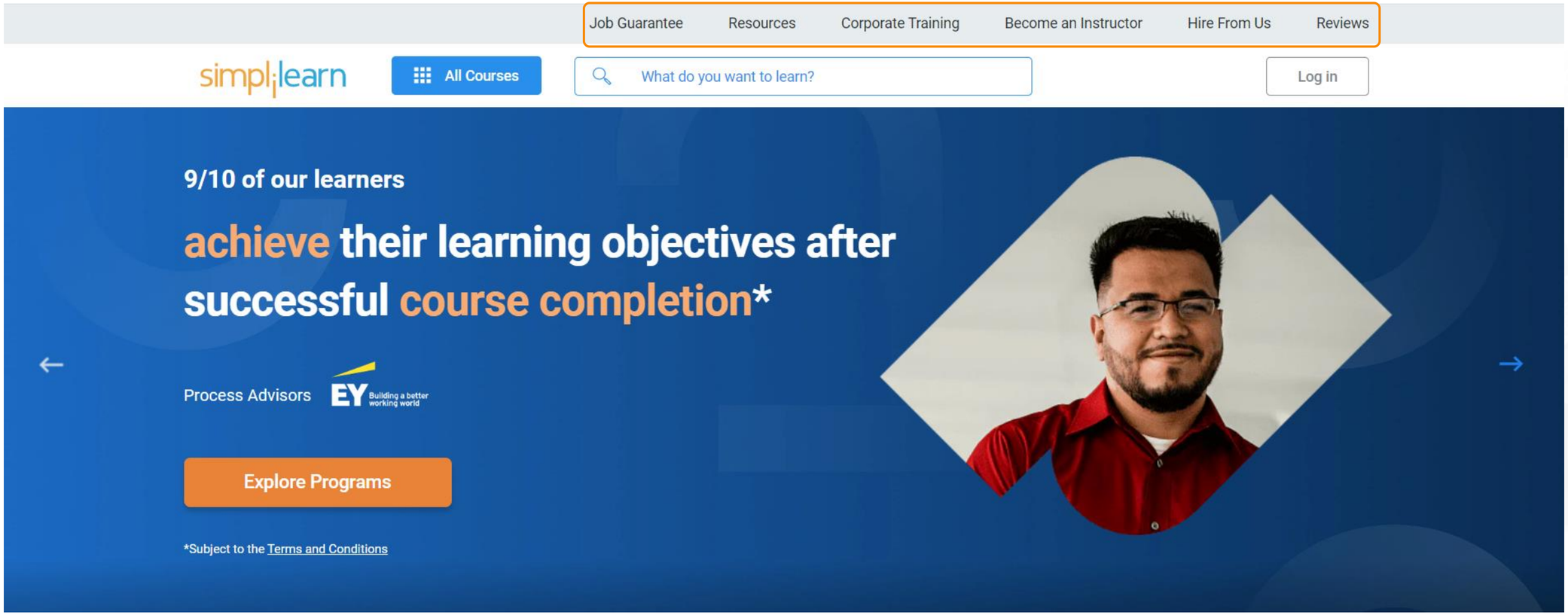
It permits different path-matching strategies.

Angular Router can interpret a browser URL as an instruction to navigate to a client-generated view.

Developers can pass optional parameters to the supporting view components that help determine the specific content to display.

Angular Routing: Example

Click on different menus to navigate to a new page



Angular Routing: Example

Case 1:
url: localhost:4200/home

AppComponent

```
<router-outlet>

        HomeComponent

</router-outlet>
```

Case 2:
url: localhost:4200/resources

AppComponent

```
<router-outlet>

        ResourcesComponent

</router-outlet>
```

Angular Route and Router

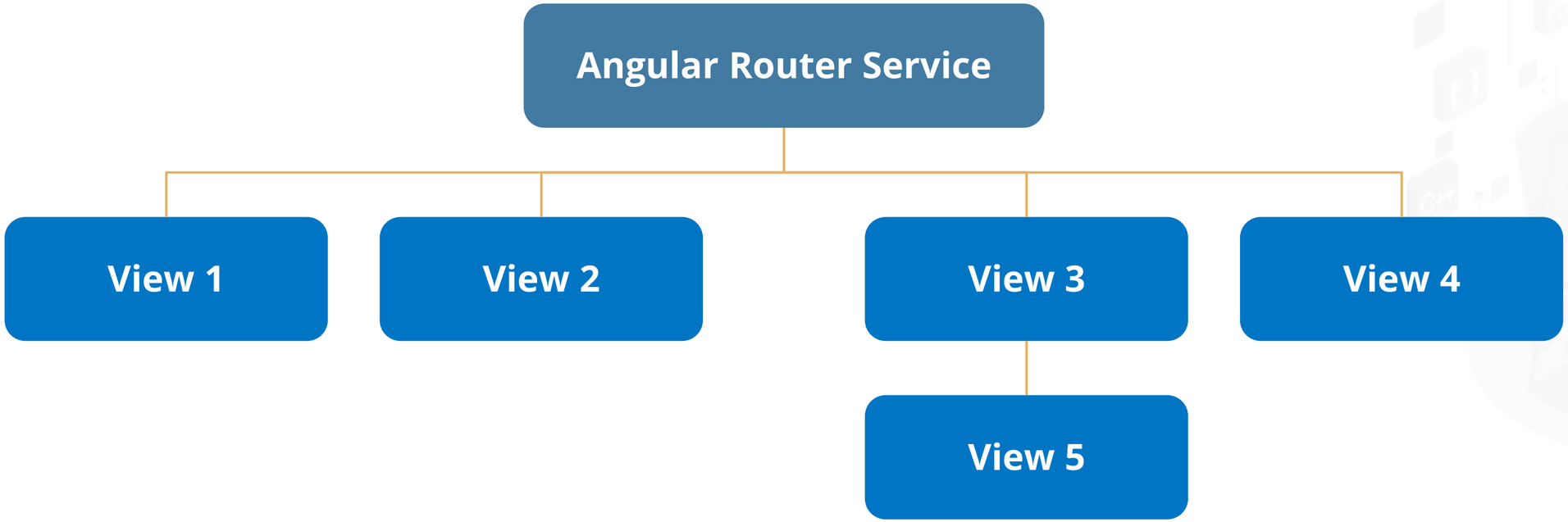
Route tells the Angular Router what to show when a user clicks a link or pastes a URL into the browser's address bar.

- Routes are an array of route objects supported by the application.
- The router object uses routes to parse and constructs the final URL.
- Each route has an associated path and component.

Routing Module

Adds directives and providers for in-app navigation among views defined in an application

Use the Angular router service to declaratively specify application states and manage state transitions.

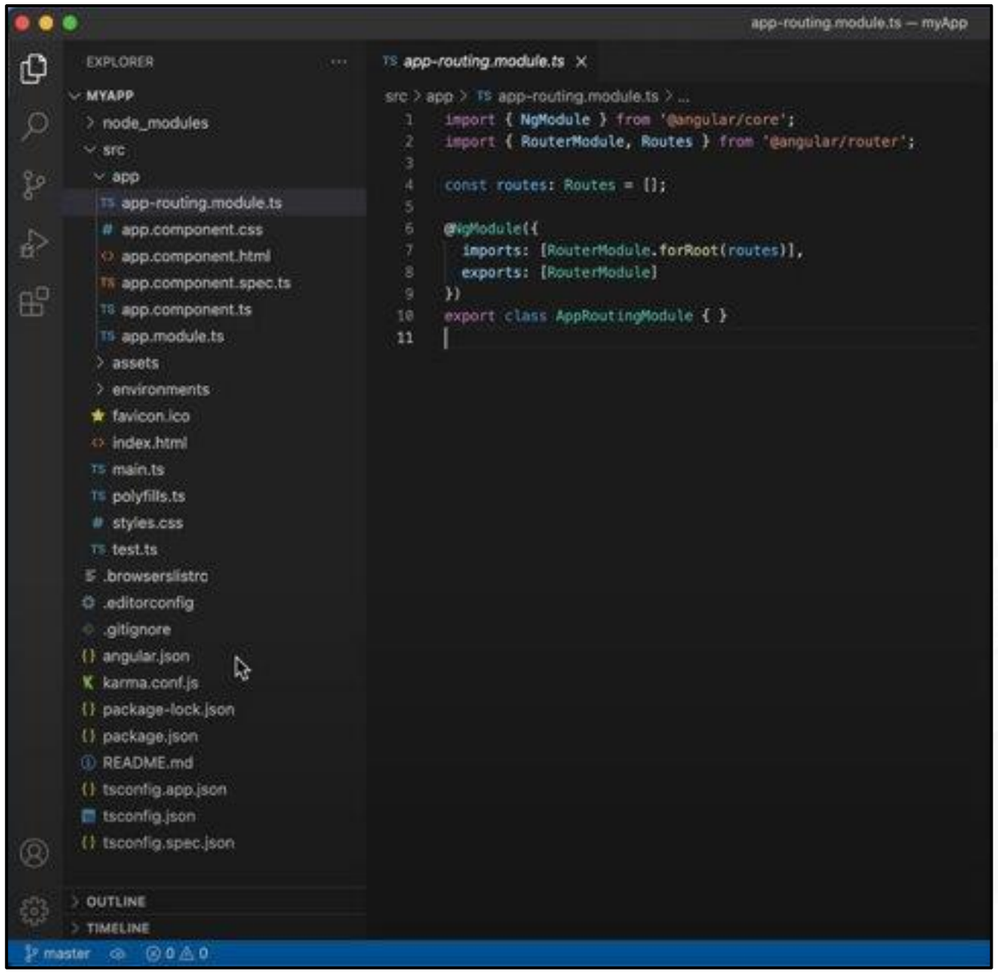


Add a Routing Module

Add routing while creating a project

```
$ ng new <project name> --routing
```

This will create an **app-routing.module.ts** file inside the src/app folder.

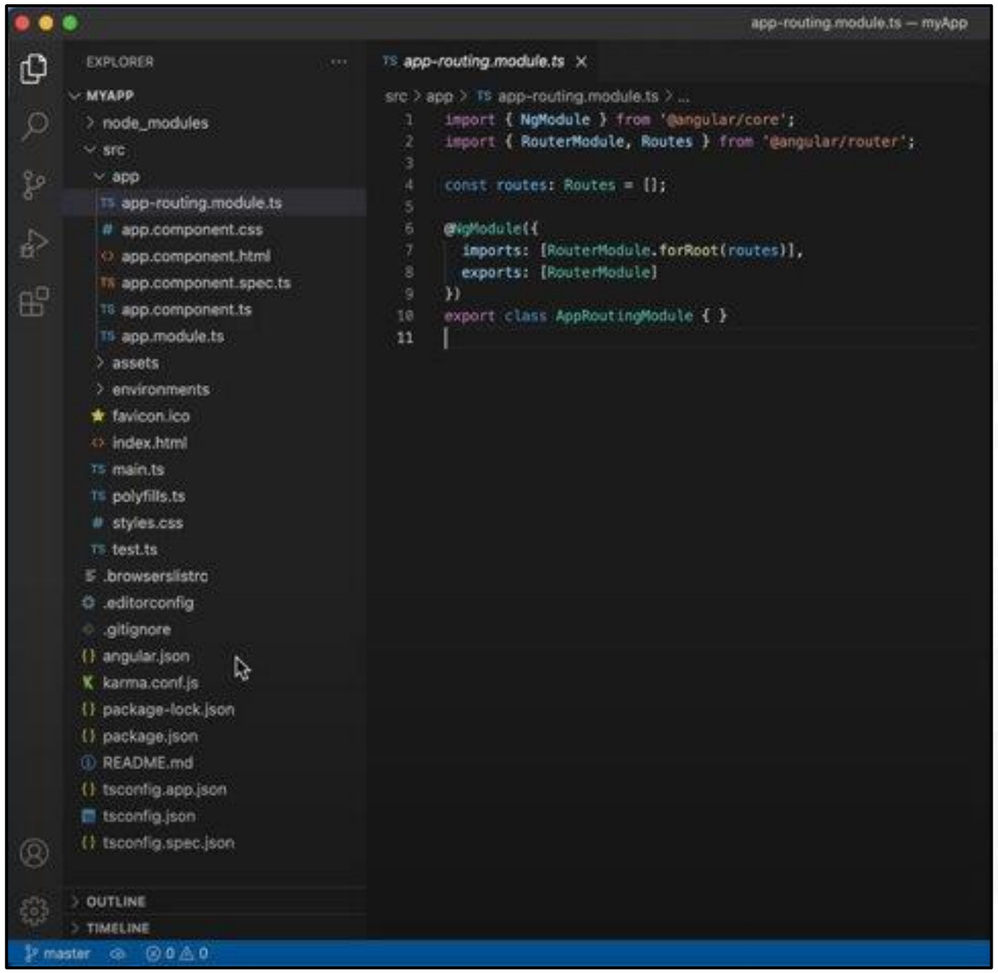


Add a Routing Module

Add routing to a module

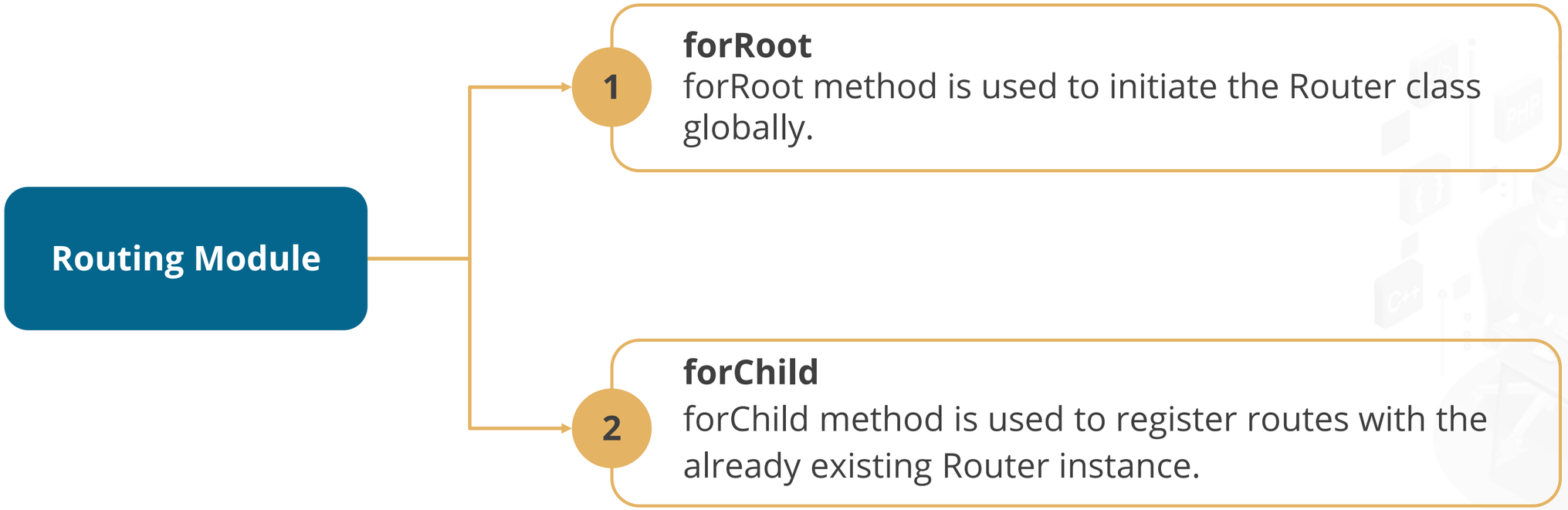
```
$ ng g module <module name> --routing
```

This will create **<module name>-routing.module.ts** file inside the newly created module folder.



Routing Module: Methods

forRoot and **forChild** are two static methods that come with Routing Module.



Basic Structure of Routing Module

NgModule is an Angular-provided class that is used to define modules in the application via the @NgModule decorator.

RouterModule is used to register routes in the application.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from
'@angular/router';
const routes: Routes = [];

@NgModule({
imports: [RouterModule.forRoot (routes)],
exports: [RouterModule]
})

export class AppRoutingModule }
```

Basic Structure of Routing Module

RouterModule has a special method called **forRoot()** that can be used to register a module or application root.

RouterModule provides all the directives and service providers required for routing.

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from
'@angular/router';
const routes: Routes = [];

@NgModule({
imports: [RouterModule.forRoot (routes)],
exports: [RouterModule]
})

export class AppRoutingModule }
```

Basic Structure of Routing Module

Routes contain an array of route objects.

Each of the objects mainly comprises two properties:

1 Path

It refers to the URL path for routing.

2 Component

It refers to the component which will be used for the corresponding path.

Adding Routes in the Routes Constant

- Adding routes creates a constant variable of type "Routes" to store the JavaScript object
- Each JavaScript object specifies a root and a component to load into that root.

RouterModule is used to register routes in the application and provides all the directives and service providers required for routing.

```
import { CustomerComponent } from ./customer/
customer.component'; import { ProductComponent }
from './product/product.component';

const routes: Routes = [

{path: 'customer', component:
CustomerComponent},
{path: 'product', component: ProductComponent},
{path: ', component: CustomerComponent}
];
```

Import Routing Module to Main Module

AppRoutingModule must be imported into the AppModule by the user.

Importing the AppRoutingModule into the AppModule

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

RouterOulet and RouterLink

RouterOutlet: A directive where the router views are displayed

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

RouterLink: A directive that is bound to an HTML element

```
<button type="button”
[routerLink]="[‘../list’]” >List</button>
```

Adding Routes

Step 1 Set the app's routes to an array of objects and pass them to Angular's **RouterModule.forRoot**

```
import { RouteModule, Routes } from '@angular/router';
const routes: Routes = [
 { path: 'component-one', component: ComponentOne }, { path: 'component-two', component:
ComponentTwo}
 ];
export const routing = RouterModule.forRoot(routes);
```

File name: app.routes.ts

Adding Routes

Step 2 Import the routing configuration into the application's root directory

```
import { routing } from './app.routes';
@NgModule({
imports: [ BrowserModule, routing], 4
 declarations: [
  AppComponent,
 ComponentOne,
 ComponentTwo
 ],
bootstrap: [ AppComponent ]
})
 export class AppModule {}
```

Adding Routes

Step 3 Redirect the application to one of the named routes. By default, the application goes to an empty path.

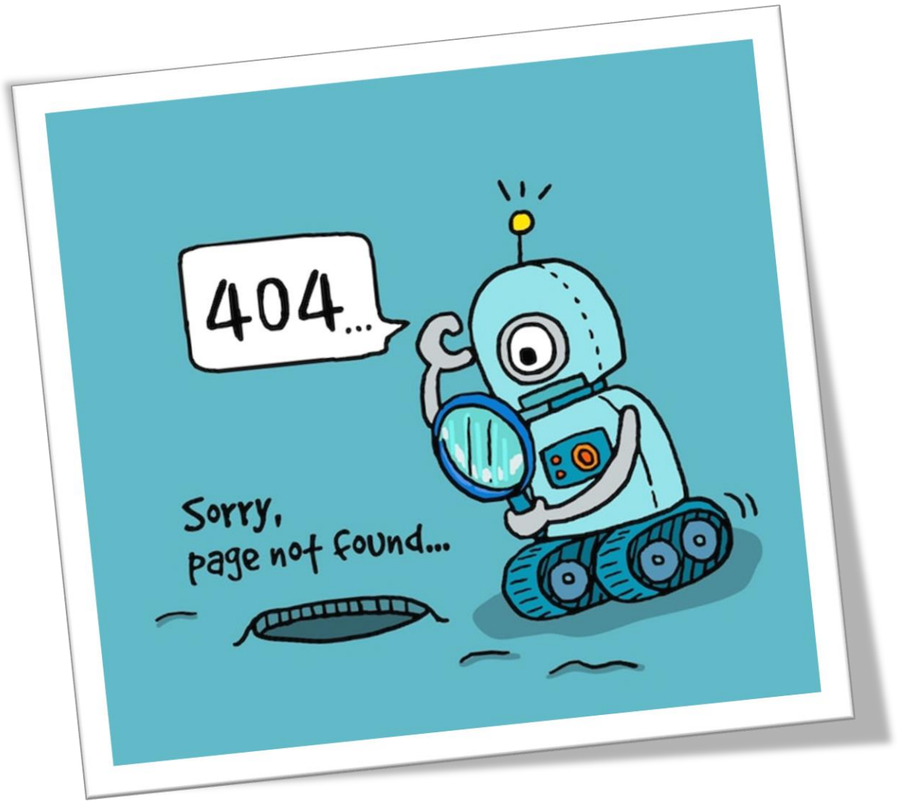
```
{path: '', redirectTo: 'component-one', pathMatch: 'full'}
```

If the full URL matches the empty path (''), the path will be changed to component-one.

pathMatch attribute required for redirection tells the route how to match the provided URL to redirect to the specified route.

Wildcard Routes

Wildcard routes are primarily used in Angular applications to handle invalid URLs.



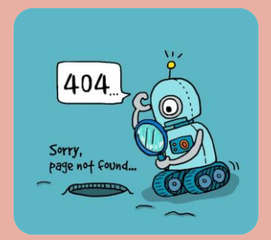
If the user enters an invalid URL or removes the real URL from the app, it defaults to a 404 error page.

Setup Wildcard Routes

A good application should be able to easily handle the user's attempts to navigate to non-existent parts of the application.

```
{ path: '**', component: <component-name> }

const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '**', component: PageNotFoundComponent },
];
```



Wildcard route for a 404 error page

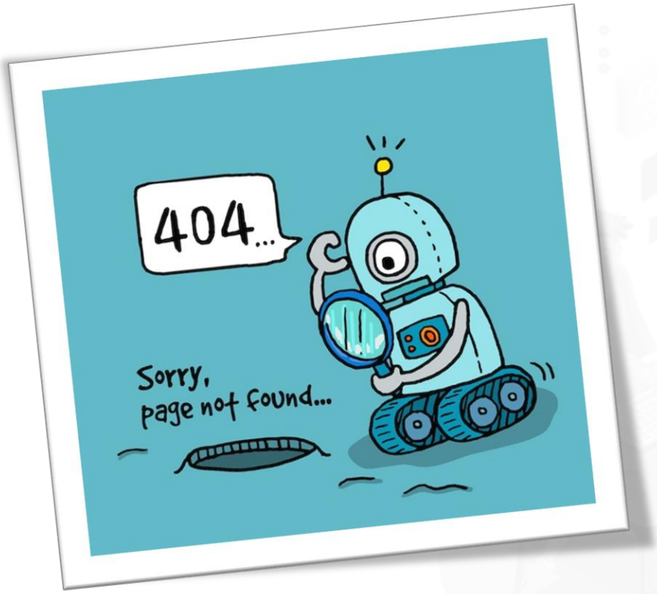
The last route with ** path is considered as a wildcard route.

Define 404 Page

The user needs to define a 404 error page, as the server does not handle the 404 error page in the Angular app.

```
const routes: Routes = [
{path:'',component:HomeComponent},
{path:'about-us',component:AboutUsComponent},
{ path: '**', component:ErrorPageComponent}
];
```

Use double asterisks in the route



Redirecting Routes

A developer can configure the router to redirect to named routes by default.

- When the application starts, it defaults to an empty path.
- The redirect path converts the original relative URL ("") into an applicable base path.

Setup Redirect Route

To set up a redirect, developer needs to set the route with a pathMatch.

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' },// redirect to `first-
component`
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

path: '' means use a starting relative URL ('').

Setup Redirect Route

To set up a redirect, developer needs to set the route with a pathMatch.

```
const routes: Routes = [
  { path: 'first-component', component: FirstComponent },
  { path: 'second-component', component: SecondComponent },
  { path: '',   redirectTo: '/first-component', pathMatch: 'full' },// redirect to `first-b
component`
  { path: '**', component: PageNotFoundComponent },  // Wildcard route for a 404 page
];
```

- Tells the router which path to redirect to, which component to redirect to, and how to match the URL
- This redirect takes precedence over wildcard routes.

The routerLinkActive Directive

routerLinkActive directive is used with the **routerLink** directive.

Template expression: The template expression must contain a space-separated string of CSS classes that will be applied to the element when the route is activated.

```
<ul class="nav navbar-nav">
 <li><a routerLink="/" routerLinkActive="active"
[routerLinkActiveOptions]="{ exact: true }">Home</a></li>
 <li><a routerLink="/about" routerLinkActive="active" >About</a></li>
<li><a routerLink="/contact" routerLinkActive="active" >Contact</a></li>
</ul>
```

Navigation

Navigation

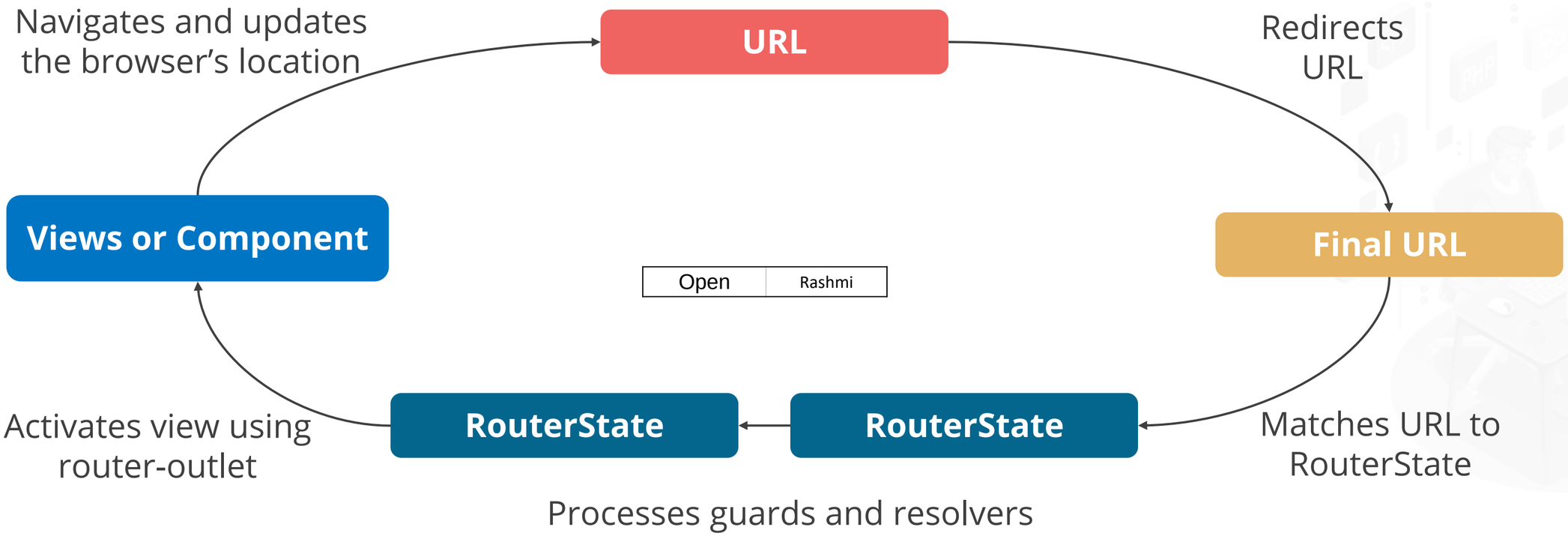
The process of determining which representation matches a navigation element is called **routing**.

Angular provides an extensive set of navigation features to accommodate simple and complex scenarios.

A **RouterModule** is required to set up navigation in Angular applications.

Router's Navigation Cycle

Angular router traverses the URL tree and matches the URL segments against the paths configured in the router configuration.



Router’s Navigation Cycle

In the router’s navigation cycle, the router creates the router state and instantiates the components.

Two ways to move the router state to another router state are:

01 Strictly by calling the **router.navigate**

02 Declaratively using the **RouterLink directive**.

Programmatic Navigation

Programmatic navigation uses events that take place along the path to redirect the user.

Common examples:

Signing in

Registering

Submitting a form

Programmatic Navigation: Methods

There are two methods to navigate programmatically using **React Router.**

1

<Navigate/>

Users can access navigate by importing it from the **react-router-dom package**.

2

navigate()

Users can access navigation using the **useNavigate custom hook**.

Route Parameters

Route parameters in Angular are used to pass information in the URL path to the component.

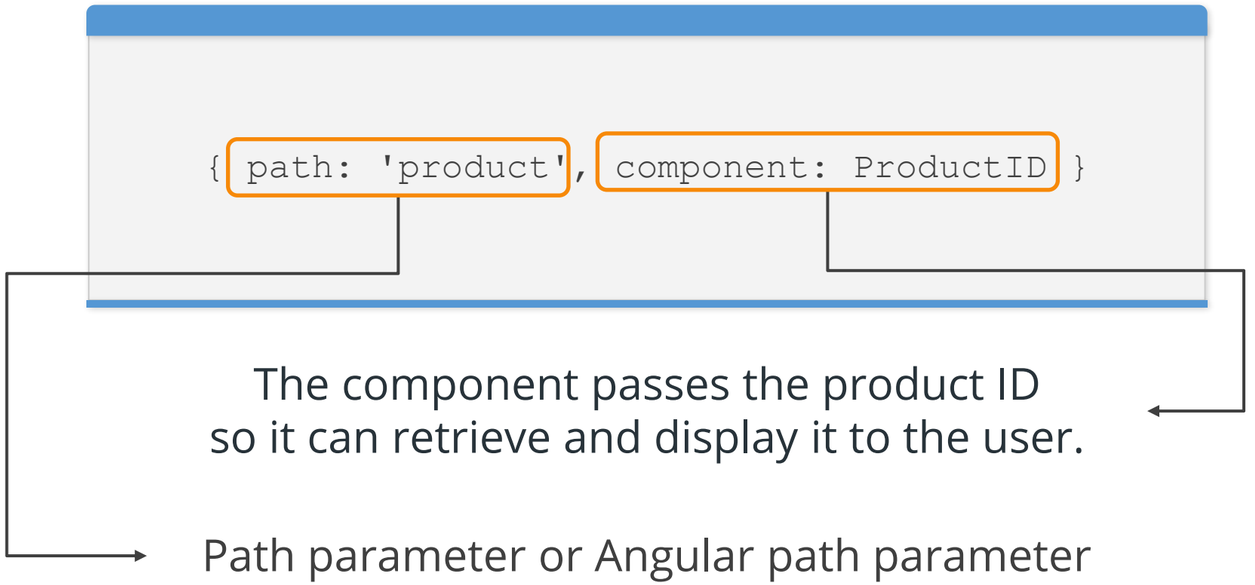
It is an effective part of a route.

It is required to define a route.

Passing Route Parameters

Scenario: Where one can use route parameters to pass the route.

Navigate to the product details view



Passing Route Parameters

```
{ path: 'product', component: ProductID }
```

- Given route will only match if the URL is **/product.**
- For retrieving the details of the product, the URL must look something like this:
  /product/1
  /product/2

Query Parameters

Query parameters are a defined set of parameters attached to the end of a URL.

A URL extension defines specific contents or actions to apply to transmitted data.

Query Parameters

Query parameters in Angular allow the construction of URLs from query strings.

Query parameters allow three optional parameters.

Page number

Sorting

Filtering criteria

Adding Query Parameters

Query parameters are not part of the path. Therefore, do not define it as a path parameter in the path array.

There are three ways to pass parameters to routes.

1. Using routerLink directive
2. Using router.navigate method
3. Using the router.navigateByUrl method

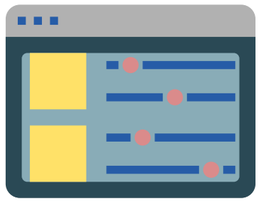
When to Use Query Parameters?

There are several aspects to consider when adding status to query parameters. The valid candidate should:

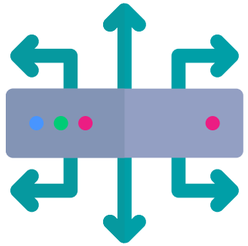
1. Affect the interface
2. Be related to user actions
3. Not be sensitive to time

Optional Parameters

These parameters allow you to pass non-compulsory parameters to routes like pagination information.



Optional parameters are a convenience feature that allows programmers to pass fewer parameters to functions and assign default values.



They allow the inclusion of information in the URL that provides hints or recommendations to the rest of the application but is not required for the application to function.

When to Use Optional Parameters?

Optional parameters are useful for model classes because values for all fields are not always available when creating a new object.

Data is fetched from the user incrementally via multiple HTML forms.

Data is generated by multiple HTTP requests to the server.

Making all constructor parameters optional allows users to construct new objects with all, some, or no data values.

Location Strategies

A location strategy describes how URLs or requests are resolved and determines the shape of the URL.

The Angular framework has a **router module** that helps to:

- Design routes and navigation links
- Develop routes and navigation links
- Implement routes and navigation links

Types of Location Strategies

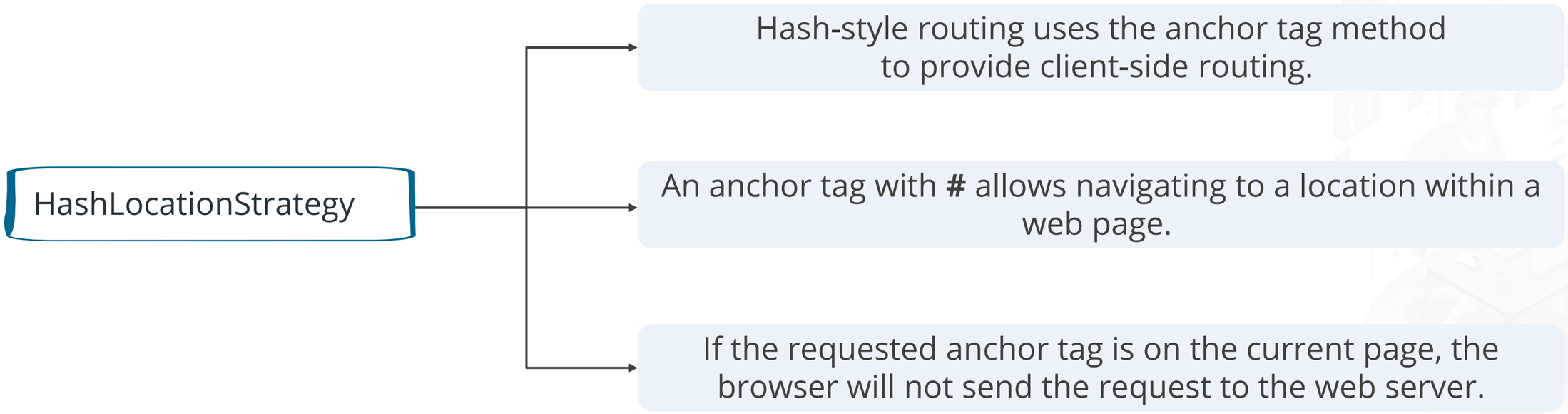
The client then handles the routing logic using the Angular router with two location strategies:

1. **HashLocationStrategy**

2. **PathLocationStrategy**

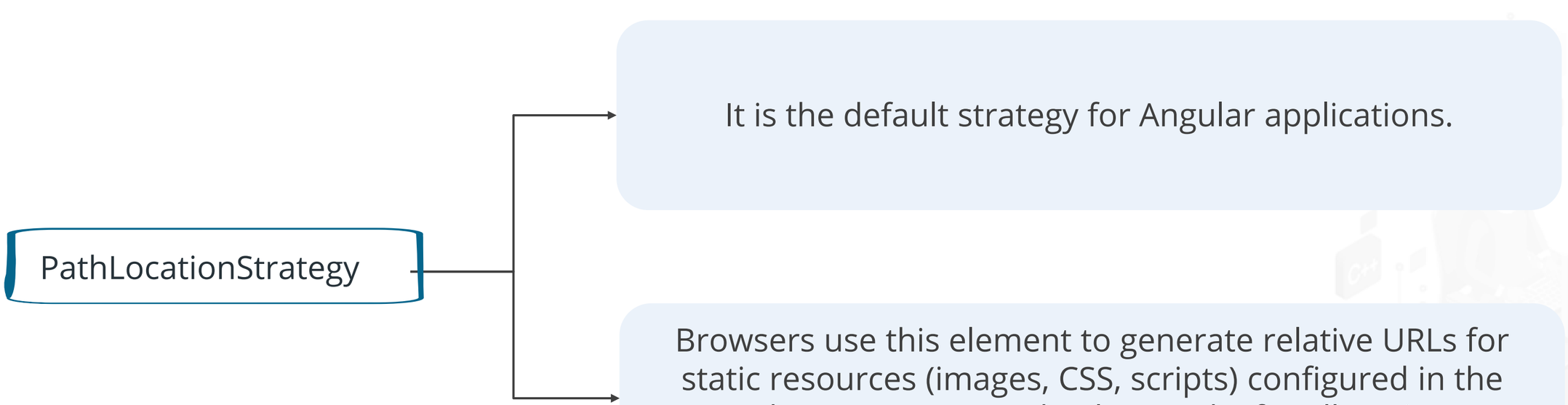
Hash Location Strategies

A **HashLocationStrategy** is used to configure location services to perform its status on hash fragments of browser URLs.



Path Location Strategies

A **PathLocationStrategy** is used to create a location service representing its state in the browser's URL path.



Choosing Location Strategy

It is recommended to use the HTML 5 style (**PathLocationStrategy**) as the location strategy due to the following reasons:

1. It creates clean, SEO-optimized URLs that are easy for users to understand and remember.

2. It takes advantage of server-side rendering, which speeds up application loading by rendering pages on the server before passing them to the client.

Use the hash location strategy only if the user needs to support older browsers.

Implementing Routing and Navigation in Angular

Duration: 25 min.

Problem Statement:

You have been assigned a task to implement routing and navigation in Angular.

Outcome:

By following the steps, you will successfully create and implement routing and navigation mechanism between web pages in Angular.

Note: Refer to the demo document for the detailed steps:
01_Routing_and_Navigation_in_Angular

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Implement Angular route and navigation
2. Execute and run the application on the browser

Accessing Routing Parameter

Duration: 25 min.

Problem Statement:

You have been assigned a task to implement routing parameters.

Outcome:

By following the steps, you will successfully create and work with Routing parameters with routerLink in Angular.

Note: Refer to the demo document for the detailed steps:
02_Accessing_Routing_Parameter

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

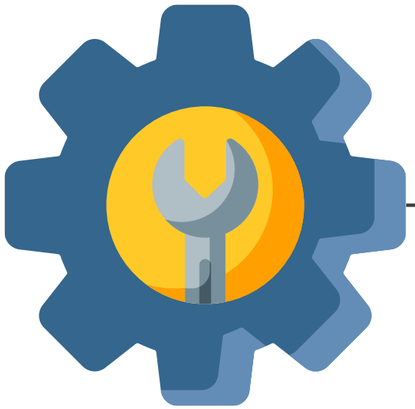
1. Implement route parameter
2. Save the file and run it on browser

TECHNOLOGY

Services and Injectables

Service

A service is a broad category that encompasses any value, function, or functionality required by an application.



Service is usually a class with a narrow and well-defined purpose

Service must be specific

Angular Service

Angular separates components from the services to grow **modularity** and **reusability**.

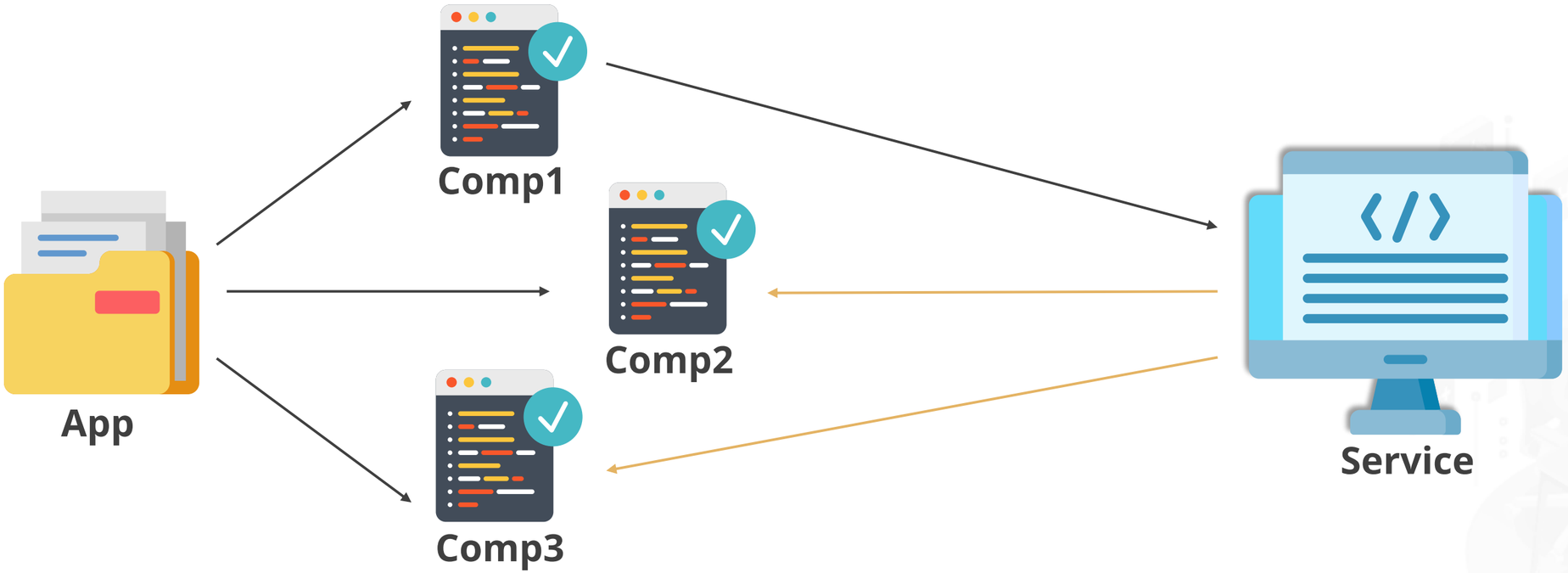
Angular services are objects that are created only once during the lifetime of the application.

They include a way to maintain data over the lifetime of the application.

The user's data is always available.

Angular services are usually implemented by dependency injection.

Angular Service



They contain methods that maintain data throughout the life of an application, that is, data is available all the time.

Features of Angular Services

These are some features of Angular services:

Exist as a class

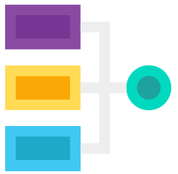
Are decorated with @injectable

Share the same piece of code

Hold business logic

Interact with backend

Share data among components

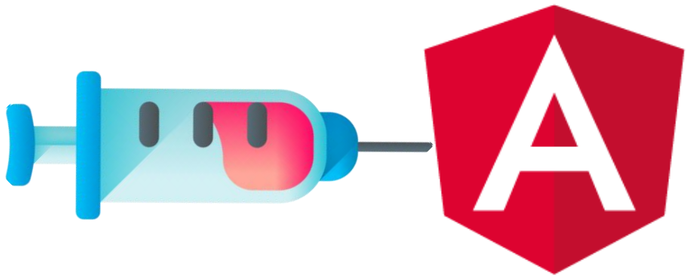


Are singleton in nature

Are registered on modules or components

Dependency Injection

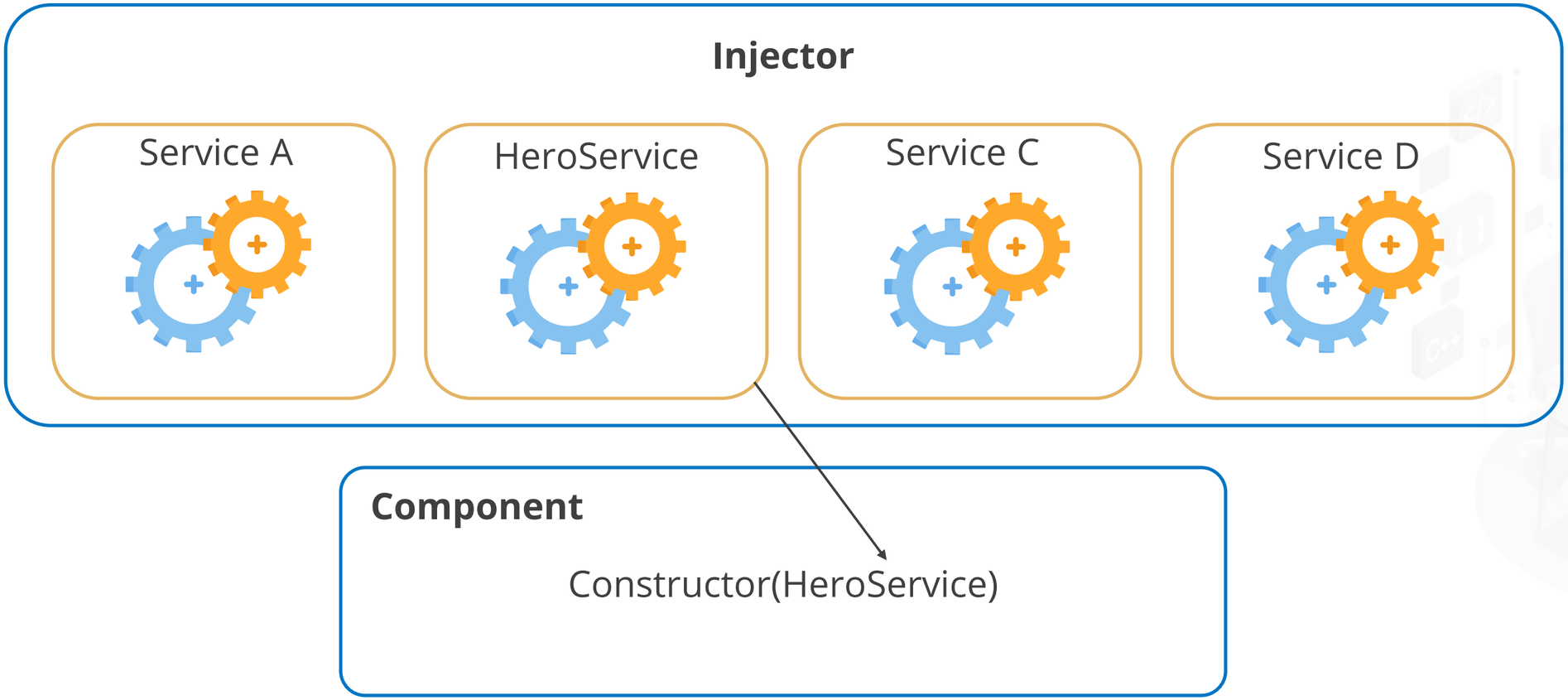
Dependency injection or DI is a design pattern and mechanism for creating parts of an application and passing them to other parts of the application that require them.



Angular supports this design pattern and applications to increase **flexibility** and **modularity**.

Service and Dependency Injection

The below image shows how service and dependency injection works:



Types of Dependency Injections

Angular has three types of dependency injections, including:

1

Constructor Injection: Users are provided with the dependencies through class constructors.

2

Setter Injection: The client uses an installation method where the injector injects dependencies.

3

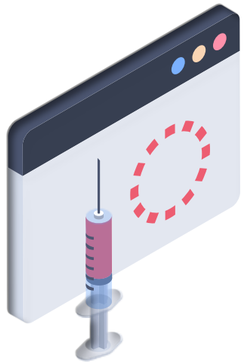
Interface Injection: Dependencies provide an injector method that injects dependencies into all forwarded clients.

Advantages of Dependency Injection

- Dependency injection helps with unit testing.
- It reduces boiler code as the injector component does dependency initialization.
- Application scaling becomes more manageable.
- It helps to provide loose coupling, which is important when programming applications.
- A dependency injection design pattern is used by Angular, which makes it very efficient.

Injectable

A class in Angular can be injected into a component as a dependency when it is defined as a service using the @Injectable() decorator.



The **@Injectable()** decorator indicates that a bean, class, channel, or **NgModule** is dependent on a service.

The injector is the main mechanism.

Injectable

Angular creates app-wide injectors during the bootstrap process and creates additional injectors as needed.

An injector creates dependencies and maintains them in a container of dependency instances.

Hierarchical Injector

A hierarchical injector system adheres to the component tree structure and enables users to specify several limits or scopes for the dependencies to run.

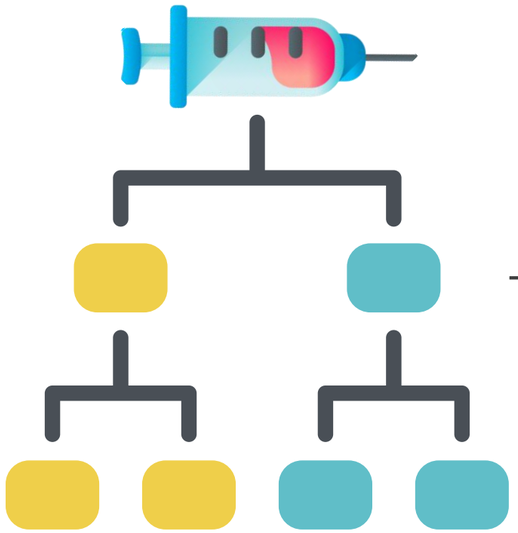
Hierarchical dependency injection allows the isolation of a section of the application.

It provides its own private dependencies that are not shared with the rest of the application.

Users can ensure that parent components share only certain dependencies with child components.

Hierarchical Injector

Angular has a hierarchical dependency injection system.

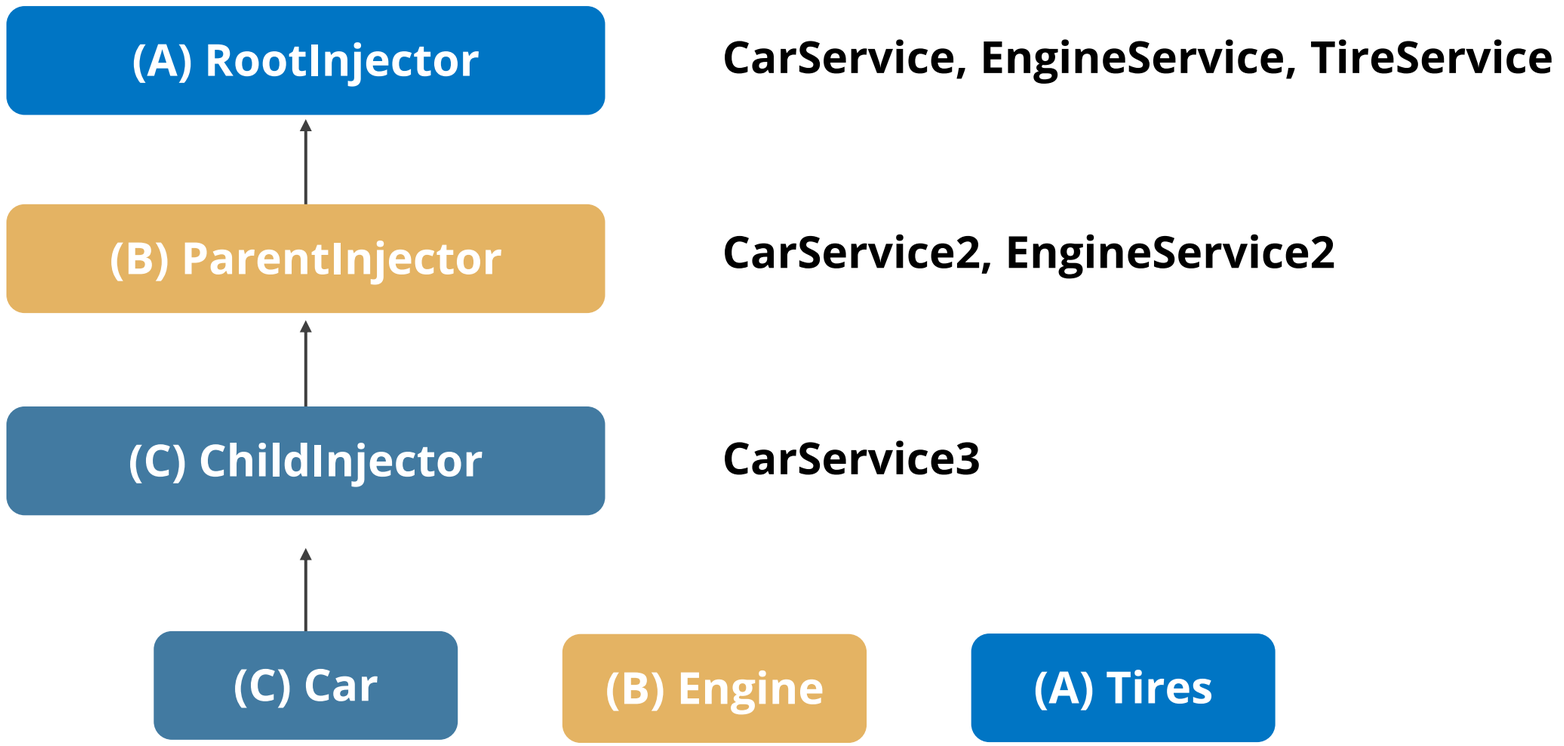


There is an injector tree parallel to the application component tree.

Injectors can be reconfigured at any level in this component tree.

Hierarchical Injector

Here is an example of the hierarchical injector:



Injecting a Service into Another Service

With the help of Angular, the user can provide component access to a service by injecting it into the component.

```
log.service.ts
 export class LogService {
   displayLog (message: string) {
              console. log (message);
              }
       }
```

- Creating a **log service** to demonstrate and understand dependency injection
- Create another service that displays a **log statement** to the console when an action is added
- Users can name this as **log service** and provide an application-level instance.

Injecting a Service into Another Service

With the help of Angular, the user can provide component access to a service by injecting it into the component.

```
app.module.ts

        import ( LogService ) from
'./services/log.service';
        ……
        @NgModule ( {
        ……
        providers: [TaskService,
LogService],
        ……
```

Instead of adding the log service inside the task create component, you can create a bean and add it as a dependency to the task service

Injecting a Service into Another Service

```
import (Injectable | from "@angular/core";
  import ( LoggingService ) from "./logging.service";
  @Injectable ()
  export class TaskService {
  …
     constructor (private logService: LoggingService)
     {
     }
  ……
  ……
  addTask (newTask: string)    {
   this.taskList.push({
     name: newTask,
      status: 'In Progress' ,
  added: new Date ()
  });
  this.logService.displayLog ( 'New Task Added...' ) ;
  }
     }
```

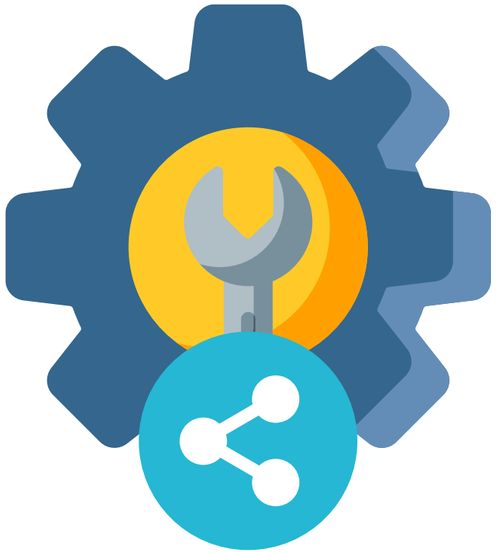
Whenever a new task is added, the log service displays the message **New Task Added**.

Services are injected into other services as dependencies.

Circular dependencies can easily occur when injecting services into other services

Service Sharing Model

A shared service provides data in a controlled manner to any component that requests it.



When sharing data, it's always a good idea to expose the state or data via methods rather than exposing data objects directly.

Using an Angular Shared Service

Angular shared service is a powerful tool for enabling communication between components.

In Angular, shared services are a publish/subscribe messaging pattern.

Typescript makes it easy to find which component references a shared service, and components make JavaScript more structured and manageable.

Using an Angular Shared Service

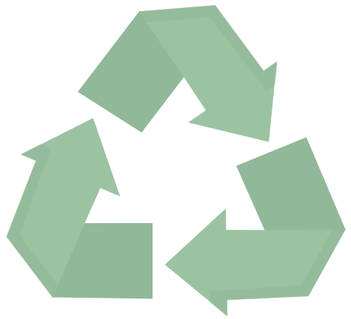
Angular shared service is a powerful tool for enabling communication between components.

If users do not use shared services in Angular, they will miss out on an important and powerful tool in the inventory.

If users have components that are not connected but need to communicate, shared services are the preferred choice.

Service Reusability

Reusable Angular services are made to contain business logic and data using various Angular components.



Service classes can be created for data or logic not associated with a specific view to be shared between components.

Reusable Angular Service: Use Cases

Reusable Angular service includes several use cases:

1. To call other APIs that return the same output format
2. To use different settings for the service

Reusable Angular Service: Use Cases

Reusable Angular service includes several use cases:

4. They are used for communication between two components.

5. Components can be in a parent-child relationship or siblings.

6. Services are used to exchange data between two components.

Using Angular Built-in Service

An **Angular** service is a feature available to the business layer of an application.

It's like a constructor function called only once using new at runtime.

Services in Angular are stateless singleton objects.

This is because only one object gets it regardless of which application interface created the service.

Types of Angular Services

There are two types of Angular services:

Built-in services

There are approximately 30 built-in Angular services.

Custom services

In Angular, users can create their own services.

Built-in Services in Angular

These are ready-made services in Angular and are automatically registered using a dependency injector at runtime.

Built-in services in Angular are:

1

$http

Service for reading data from remote servers

2

$interval

Execute function or method repeatedly with specific time intervals based on the user's requirement

3

$timeout

Allows the developer to set some time delay before the execution of the function

Built-in Services in Angular

Built-in services in Angular are:

4

$anchorscroll

Page specified by the anchor in $location.hash() is scrolled using $anchorscroll.

5

$animate

Consists of several Document Object Model (DOM) utility methods that support animation hooks

6

$animateCss

Animates if ngAnimate includes it by default.

7

$compile

Allows HTML or DOM strings to be compiled into templates and creates a template function to tie the template and scope together

Custom Services in Angular

For creating a service, connect the service to the module.

Create a service named **hexafy**

```
app.service('hexafy', function() {
  this.myFunc = function (x) {
    return x.toString(16);
  }
});
```

Custom Services in Angular

For defining the controller, add dependency as a custom-made service.

Use the custom-made service named **hexafy** to convert a number into a hexadecimal number:

```
app.controller('myCtrl', function($scope, hexafy) {
  $scope.hex = hexafy.myFunc(255);
});
```

Creating Services for the Data in the Application

Duration: 20 min.

Problem Statement:

You have been assigned a task to implement Service for data in the application.

Outcome:

By following the steps, you will successfully crate and work with services in Angular for Data manipulation.

Note: Refer to the demo document for the detailed steps:
03_Creating_Services_for_the_Data_in_the_Application

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create service for product data in the application

Implementing Service and Injectables

Duration: 20 min.

Problem Statement:

You have been assigned a task to implement Service and Injectable in Angular.

Outcome:

By following the steps, you will successfully create and work with services along with dependency injections in Angular.

Note: Refer to the demo document for the detailed steps:
04_Implementing_Service_and_Injectables

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create injectables in Angular

Routing Mechanisms

Routing Structure

Whenever the user navigates from page to page, that is, clicks a link or changes the browser's URL, the Angular router ensures the application responds appropriately.

It is a set of instructions used to specify the direction in which data packets traveling across an IP network will travel.

Routing Structure

For appropriate application response, the corner router performs the following seven steps.

Step 1: Parse the URL

Step 2: Redirect to the URL

Step 3: Identify the router state

Step 4: Run guards

Step 5: Resolve data for router state

Step 6: Activate component

Step 7: Manage navigation

RxJS Library

RxJS is a library for creating asynchronous and event-driven programs using observable sequences.

The library also provides utility functions for creating and working with observables.

https://rxjs.dev/

RxJS Library

RxJS provides operators inspired by:

- One primitive type
- Auxiliary types (observer, schedulers, subjects)
- Array methods
- Asynchronous events

Core RxJS Concepts

The core concepts in RxJS that solve async event management are:

Observable	Represents the idea of a callable collection of future values or events
Observer	Set of callbacks that know how to listen for values provided by an Observable
Subscription	Represents the execution of an Observable and is useful primarily for unsubscribing from an Observable

Core RxJS Concepts

The core concepts in RxJS that solve async event management are:

Operators	Provides a functional programming style for handling collections with operations such as matching, filtering, concatenating, and reducing
Subject	To multicast values or events to multiple Observers
Schedulers	A centralized dispatcher for concurrent control allows the coordination of calculations as they occur. For example, setTimeout or requestAnimationFrame.

Advantages of Using RxJS

- RxJS can be used with other JavaScript libraries and frameworks. It is supported in JavaScript and typescript. Some examples are Angular, ReactJS, VueJS, and NodeJS.

- RxJS is a great library when it comes to handling asynchronous operations.

Advantages of Using RxJS

RxJS makes reactive programming work by using observables to handle asynchronous data calls, callbacks, and event-driven programs.

RxJS provides a set of operators used with reactive programming in:

- Mathematical categories
- Transformations
- Filtering and utilities
- Conditional operations
- Error handling and joins

Implementing Routing Mechanisms

Duration: 20 min.

Problem Statement:

You have been assigned a task to implement static and dynamic routing.

Outcome:

By following the steps, you will successfully create and work with static and dynamic routing using routing parameters in Angular.

Note: Refer to the demo document for the detailed steps:
05_Implementing_Routing_Mechanism

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create an Angular app and perform static routing
2. Perform dynamic routing

Implementing Child and Parent Route

Duration: 25 min.

Problem Statement:

You have been assigned a task to implement child and parent route.

Outcome:

By following the steps, you will successfully implement child-parent route between various components and pages in Angular.

Note: Refer to the demo document for the detailed steps:
06_Implementing_Child_and_Parent Route

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create an Angular app
2. Run it on the browser
3. Create child and parent route
4. Check the output on browser

Rest APIs

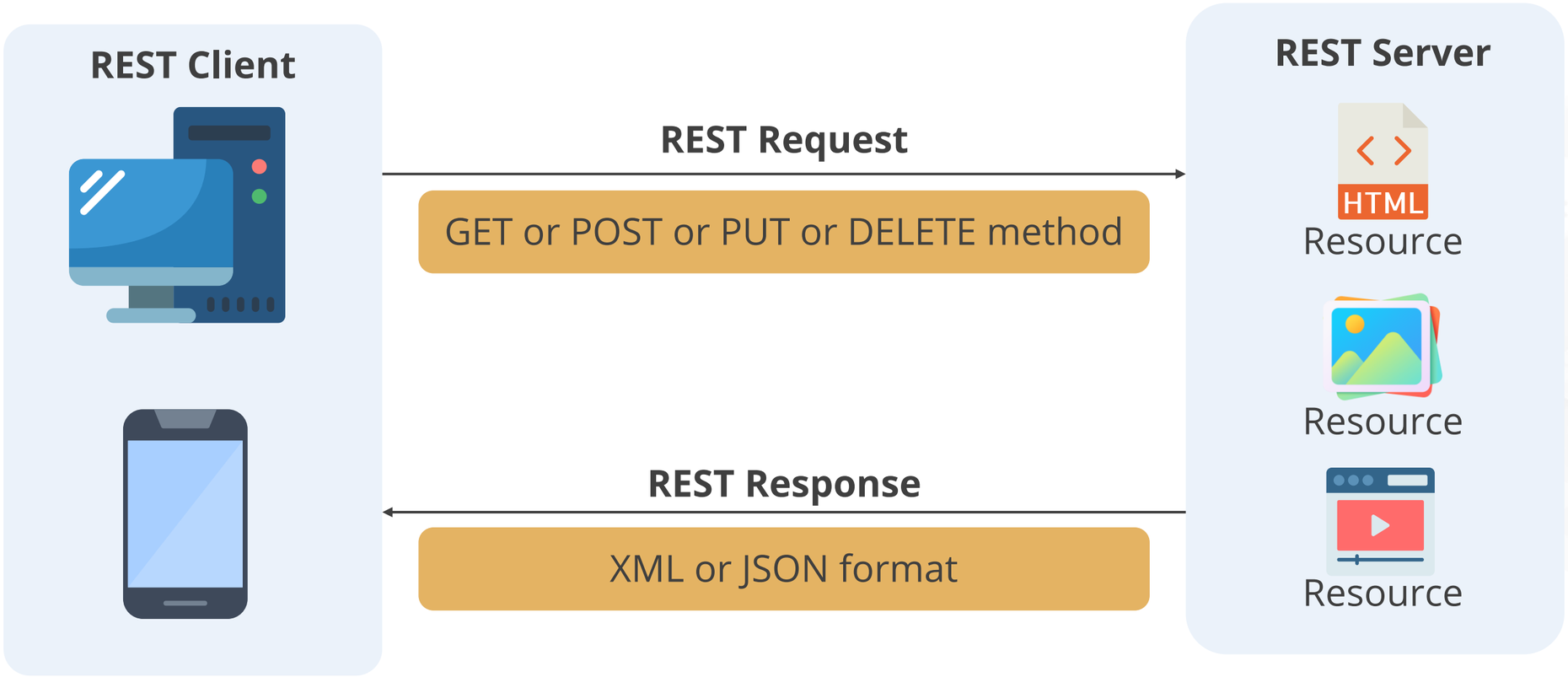
REST API calls

REST, **Representational State Transfer**, is an architectural style that defines a set of constraints used to build web services.

A REST API is an easy and flexible way to access web services without requiring any processing.

REST API Working

A call is made from a client to a server, and data is received back over the HTTP protocol.



REST API HTTP Methods

HTTP has five methods commonly used in REST-based architectures such as **GET**, **PUT**, **POST**, **DELETE**, and **PATCH**.

These are used to perform **CRUD** operations, **Create**, **Read**, **Update**, and **Delete**.

- Other less common methods include OPTIONS and HEAD. These are used to perform CRUD operations, such as **Create**, **Read**, **Update**, and **Delete**.
- There are other less common methods such as **OPTIONS** and **HEAD**.

REST API HTTP Methods

There are five methods that are frequently used in a REST API that uses HTTP.

GET: Used to read (or retrieve) representations of resources

POST: Used commonly to create new resources and used to create child resources

PUT: Used to update the potentialities

PATCH: Used to modify abilities

DELETE: Used to delete a resource that is identified by URL

Key Takeaways

- Angular Router can interpret a browser URL as a instruction to navigate to a client-generated view.
- Angular provides an extensive set of navigation features to accommodate simple scenarios to complex scenarios.
- A service is a broad category that encompasses any value, function, or functionality required by an application.
- An Angular service is an object that is, instantiated only once during the application’s lifetime.

TECHNOLOGY

Thank You

simplilearn