Routes in Angular:

1. Introduction to Routing:

- Routing in Angular refers to the navigation mechanism that allows users to move between different components in an Angular application.
- The Angular Router provides a powerful way to define navigation paths and control the flow of the application.

2. Setting Up Routes:

- To enable routing in an Angular application, you need to import the `RouterModule` and define routes in the `RouterModule.forRoot()` method.

```
// app.module.ts
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
];

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

3. Router Outlet:

- The `<router-outlet></router-outlet>` directive is used in the template to specify where the routed component should be displayed.

```
<!-- app.component.html -->
<router-outlet></router-outlet>
```

4. Router Links:

- Use the `routerLink` directive to create links for navigating to different routes.

```
<!-- app.component.html -->
<a routerLink="/">Home</a>
```

```
<a routerLink="/about">About</a>
<a routerLink="/contact">Contact</a>
```

5. Route Parameters:

- Routes can have parameters that allow dynamic values in the URL.

```
// app.module.ts
const routes: Routes = [
  { path: 'user/:id', component: UserProfileComponent },
];
```

```
<!-- app.component.html -->
<a [routerLink]="['/user', userId]">User Profile</a>
```

```
// user-profile.component.ts
ngOnInit() {
  this.route.params.subscribe((params) => {
    this.userId = params['id'];
  });
}
```

6. Nested Routes:

- You can nest routes within other routes to create a hierarchy of components.

```
// app.module.ts
const routes: Routes = [
  {
    path: 'dashboard',
    component: DashboardComponent,
    children: [
      { path: 'stats', component: StatsComponent },
      { path: 'reports', component: ReportsComponent },
    ],
  },
];
```

```
<!-- dashboard.component.html -->
<router-outlet></router-outlet>
```

7. Router Guards:

- Guards are used to control navigation based on certain conditions. There are different types of guards:
  - **CanActivate:** Determines if a route can be activated.
  - **CanDeactivate:** Determines if a route can be deactivated.
  - **CanLoad:** Prevents the lazy-loaded module from being loaded prematurely.
  - **Resolve:** Fetches data before the route is activated.

```
// auth.guard.ts
canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): boolean
{
  if (this.authService.isAuthenticated()) {
    return true;
  } else {
    this.router.navigate(['/login']);
    return false;
  }
}
```

8. Lazy Loading:

- Lazy loading allows you to load modules on-demand, improving initial page load performance.

```
// app.module.ts
const routes: Routes = [
  { path: 'admin', loadChildren: () => import('./admin/admin.module').then((m)
=> m.AdminModule) },
];
```

9. Route Events:

- The `Router` service provides events that you can subscribe to, such as `NavigationStart`, `NavigationEnd`, `NavigationError`, etc.

```
// app.component.ts
ngOnInit() {
  this.router.events.subscribe((event) => {
    if (event instanceof NavigationStart) {
      // Do something on navigation start
    }
  });
}
```

10. Wildcard Route and Redirects:

- The wildcard route (`**`) can be used to handle unknown routes or create a default route.
- Redirects allow you to redirect from one route to another.

```
// app.module.ts
const routes: Routes = [
  { path: '404', component: NotFoundComponent },
  { path: '**', redirectTo: '/404' },
];
```

11. Location Strategies:

- Angular supports two location strategies for routing: `PathLocationStrategy` and `HashLocationStrategy`.
- The default is `PathLocationStrategy` which uses the HTML5 history API for clean URLs.

```
// app.module.ts
@NgModule({
  imports: [RouterModule.forRoot(routes, { useHash: true })],
})
```

These concepts cover the fundamentals of routing in Angular. Routing is a powerful feature that enables the creation of single-page applications with multiple views and navigation between them.

Additional Routing Concepts:

1. Route Resolvers:

- Route resolvers allow you to fetch data before activating a route. This is useful when you need to ensure that data is available before rendering a component.

```
// data-resolver.service.ts
resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
Observable<any> {
  return this.dataService.getData();
}
```

```
// app.module.ts
const routes: Routes = [
  {
    path: 'profile',
    component: UserProfileComponent,
    resolve: { userData: DataResolverService },
  },
];
```

```
// user-profile.component.ts
ngOnInit() {
  this.userData = this.route.snapshot.data['userData'];
}
```

2. Angular Route Reuse Strategy:

- By default, Angular destroys the component when navigating away from it. You can implement a route reuse strategy to cache and reuse components.

```
// route-reuse-strategy.service.ts
shouldDetach(route: ActivatedRouteSnapshot): boolean {
  return false;
}

store(route: ActivatedRouteSnapshot, handle: DetachedRouteHandle | null): void
{
  this.routeHandles.set(route.routeConfig!.path!, handle);
}

shouldAttach(route: ActivatedRouteSnapshot): boolean {
  return false;
}

retrieve(route: ActivatedRouteSnapshot): DetachedRouteHandle | null {
  return this.routeHandles.get(route.routeConfig!.path!) || null;
}
```

```
// app.module.ts
providers: [
  { provide: RouteReuseStrategy, useClass: CustomReuseStrategyService },
],
```

3. Router State Snapshot and URL Tree:

- The `RouterStateSnapshot` provides a snapshot of the router state at the current point in time. It includes information about the current URL, route parameters, and other related data.

```
// some.component.ts
ngOnInit() {
  const snapshot: RouterStateSnapshot = this.router.routerState.snapshot;
  const url: string = snapshot.url;
  const params: Params = snapshot.root.queryParams;
}
```

- The `UrlTree` represents the current URL as a tree structure. You can navigate and manipulate the URL using the `Router` service.

```
// some.component.ts
navigateToSubpath() {
  const urlTree: UrlTree = this.router.createUrlTree(['/parent', 'child'], {
queryParams: { key: 'value' } });
  this.router.navigateByUrl(urlTree);
}
```

4. Router Testing:

- When testing components with routing, you can use the `RouterTestingModule` to configure a test environment for routing.

```
// some.component.spec.ts
TestBed.configureTestingModule({
  imports: [RouterTestingModule.withRoutes(routes)],
});
```

```
// some.component.spec.ts
it('should navigate to the profile page', fakeAsync(() => {
  router.navigate(['/profile']);
  tick();
  expect(location.path()).toBe('/profile');
}));
```

These additional concepts provide more depth and flexibility when working with routing in Angular. Depending on the complexity of your application, you might find these features valuable for handling various scenarios.

Adding routes to an Angular project involves a series of steps. Below are detailed steps to set up and configure routes in an Angular project

Step 1: Create Angular Project

If you haven't already created an Angular project, you can do so using the Angular CLI:

```
ng new your-project-name
cd your-project-name
```

Step 2: Create Components

Create the components that you want to navigate to. For example, let's create three components: `HomeComponent`, `AboutComponent`, and `ContactComponent`:

```
ng generate component home
ng generate component about
ng generate component contact
```

Step 3: Set Up Routes in `app.module.ts`

Open the `app.module.ts` file and set up the routes using the `RouterModule`. Specify the base URL using the `<base>` tag in `index.html`.

`index.html`:

```
<!-- index.html -->
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Your Angular App</title>
  <base href="/your-base-url/">
  <!-- Other head elements... -->
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

`app.module.ts`:

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { RouterModule, Routes } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'contact', component: ContactComponent },
];

@NgModule({
  declarations: [
    HomeComponent,
    AboutComponent,
    ContactComponent,
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot(routes),
  ],
  bootstrap: [AppComponent],
})
export class AppModule { }
```

Step 4: Set Up Router Outlet in `app.component.html`

Open the `app.component.html` file and add the `<router-outlet></router-outlet>` where you want the routed components to be displayed:

```
<!-- app.component.html -->
<div>
  <h1>My Angular App</h1>
  <nav>
    <a routerLink="/">Home</a>
    <a routerLink="/about">About</a>
    <a routerLink="/contact">Contact</a>
  </nav>
</div>

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

Step 5: Run the Application

Run your Angular application using the following command:

```
ng serve
```

Visit `http://localhost:4200/your-base-url/` in your browser, and you should see your Angular app with navigation links.

Step 6: Test Navigation

Click on the navigation links to test if the routing is working. The base URL `/your-base-url/` will serve as the starting point for all routes.

By following these steps, you have specified a base URL for your Angular application, and all the routes will be relative to this base URL. Adjust the base URL according to your deployment environment.

pathMatch

In Angular routing, the `pathMatch` property is used to specify how the router should match the URL against the defined routes. The `pathMatch` property is part of the `Route` object and can take one of the following values:

1. `'prefix'`: (default) Matches if the URL starts with the path specified in the route.
2. `'full'`: Matches if the URL matches the entire path specified in the route.

Here's how you can use the `pathMatch` property in your routes:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent, pathMatch: 'full' },
  { path: 'about', component: AboutComponent, pathMatch: 'prefix' },
  // Other routes...
];
```

Examples:

1. `'prefix'` (default):

Assume you have the following route:

```
{ path: 'about', component: AboutComponent }
```

- URL: `/about` matches.
- URL: `/about/contact` matches (because it starts with `/about`).

2. `'full'`:

```
{ path: 'home', component: HomeComponent, pathMatch: 'full' }
```

- URL: `/home` matches.
- URL: `/home/contact` does not match (because it doesn't match the entire path).

When to Use:

- **`'prefix'`:** Use when you want a route to match if the URL starts with the specified path.
- **`'full'`:** Use when you want a route to match only if the entire URL matches the specified path.

Default Behavior:

If `pathMatch` is not specified, it defaults to `'prefix'`. This means that routes will match if the URL starts with the specified path.

```
// Defaults to pathMatch: 'prefix'
{ path: 'home', component: HomeComponent }
```

Both `'prefix'` and `'full'` have their use cases, and the choice depends on the desired behavior of your application.

wildCard

In Angular routing, a wildcard route is used to handle URLs that do not match any of the defined routes. The wildcard route is denoted by using the `**` (double asterisk) as the path. It is also known as a catch-all route because it catches any URL that does not match the defined routes.

Here's an example of how to use a wildcard route:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  // Other routes...

  // Wildcard route
  { path: '**', component: NotFoundComponent } // Redirect to a "Not Found"
component
];
```

In this example:

- If the URL is `/home`, it matches the first route and renders the `HomeComponent`.
- If the URL is `/about`, it matches the second route and renders the `AboutComponent`.
- If the URL does not match any of the defined routes (e.g., `/nonexistent`), it matches the wildcard route and renders the `NotFoundComponent` or any other component you specify.

Use Cases:

1. **Handling 404 Errors:** A wildcard route is often used to handle 404 errors, providing a user-friendly "Not Found" page when the requested URL doesn't match any known route.

2. **Fallback Route:** It can also be used as a fallback route when you want to redirect to a default component for any unmatched URL.

Example with Redirect:

If you want to redirect unmatched routes to a specific route, you can use the wildcard route along with a redirect:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  // Other routes...

  // Redirect unmatched routes to the home page
  { path: '**', redirectTo: 'home' }
];
```

In this example, any URL that doesn't match a specific route will be redirected to the `home` route.

Keep in mind that the wildcard route should typically be the last route in your configuration, as it will match any URL, and subsequent routes won't be reached if the wildcard route is placed earlier.