

# ROUTING

Ori Calvo, 2017

[oric@trainologic.com](mailto:oric@trainologic.com)

<https://trainologic.com>

# Routing

2

- Simulate a “standard” browser navigation inside SPA
- A routing table describes the association between URLs and components
- Integrate with browser’s history buttons
- Navigation is done using links or code (imperatively)

# Why bother changing the URL ?

3

- Keeping the same URL for all views means
  - ▣ F5 resets current view
  - ▣ Bookmarking is lost
  - ▣ Back/forward buttons are disabled
  - ▣ Sharing between users is limited
  - ▣ Not what the user expects

# The Recipe

4

- ❑ base tag
- ❑ Install `@angular/router` and import it
- ❑ Define routes
- ❑ Define outlet
- ❑ Use links
- ❑ Use router state inside components/services

# Base tag

5

- Application may be served using URL
  - <http://domain/index.html>
- Or
  - <http://domain/app/index.html>
  - A.K.A virtual directory
- The latter requires that Angular knows the base address of the application in order to generate correct URLs
- Use `<base href="/app" />` inside index.html

# angular/@router

6

- This is not the only available router for Angular
  - ▣ See **ui-router**
- `yarn add angular/@router`

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

Let angular  
know the routing  
table definition

# Define routes

7

- Each route consists of
  - Url
  - Component
  - Additional options

```
export const routes: Routes = [  
  {  
    path: "",  
    component: HomeComponent,  
    pathMatch: 'full'  
  },  
  {  
    path: '**',  
    redirectTo: "",  
  },  
];
```

The root component is not part of the routing table

# Routes

8

- ❑ Each route map a URL to a single component
- ❑ No leading slash
- ❑ The order does matter
- ❑ First matching route win !!!
- ❑ Specificity does not matter 😞
- ❑ Nested routes are supported



# The outlet

9

```
<h1>Hello AngularJS</h1>  
  
<router-outlet></router-outlet>
```

Inside app.component.html  
all components are injected  
after router-outlet

```
<my-app _ngghost-c0="" ng-version="4.1.1">  
  <h1 _ngcontent-c0="">Hello AngularJS</h1>  
  <router-outlet _ngcontent-c0=""></router-outlet>  
  <my-home _ngghost-c1="">  
    <h1 _ngcontent-c1="">Home</h1>  
  </my-home>  
</my-app>
```

DOM at runtime. my-  
home is injected after  
the outlet (not inside !!!)

# Router links

10

- Don't use link with href since it causes full page reload

```
<div class="site-menu">  
  <a href="">Home</a>  
  <a href="about">About</a>  
</div>
```

Causes full  
page reload

- Use link with **routerLink** attribute

A directive which takes  
control over the link and  
does not let it reload

```
<div class="site-menu">  
  <a routerLink="">Home</a>  
  <a routerLink="about">About</a>  
</div>
```

# RouterLinkActive directive

11

- Assume a site menu
- It contains multiple links. Each link navigates to different view
- Usually, you want to indicate which link is the active one

```
<div class="site-menu">  
  <a routerLink="" routerLinkActive="x">Home</a>  
  <a routerLink="about" routerLinkActive="y">About</a>  
</div>
```

Will append a CSS class named "x" to the a element

# Required Parameter

12

- Use `:` syntax inside route definition

```
{  
  path: 'contact/:id',  
  component: ContactComponent,  
}
```

- Add id value to the link

```
<a [routerLink]="['contact', 1]">Contact</a>
```

- Or

```
<a routerLink="contact/1">Contact</a>
```

- But the following is not valid

```
<a [routerLink]="['contact', {id: 1}]">Contact</a>
```

# Reading parameters

13

- Use **ActivatedRoute** provider
  - ▣ Most of its properties are Observable which means you can react to URL changes
  - ▣ More details later
- Use **ActivatedRoute.snapshot** for a simple read

```
export class ContactComponent {  
  id: number;  
  
  constructor(private activatedRoute: ActivatedRoute) {  
    this.id = this.activatedRoute.snapshot.params.id;  
  }  
}
```

# Optional Parameter

14

- Each route can have its own optional parameters
- The syntax is a bit weird

```
<a [routerLink]="['contact', 1, {more: true}]">Contact</a>
```



```
http://localhost:8080/contact/1;more=true
```

- Please note that “normal” query parameters are considered global for all routes

# Optional Parameters

15

- Are not preserved when navigating between routes
- Even when navigating to the same route
- Optional and required parameter may have the same name
  - ▣ In that case the optional parameter has higher precedence which reflects inside the **ActivatedRoute.data** field
  - ▣ See more details later

# Persist Preferences

16

- Assume a component that supports a “show/hide more” button
- Each time the button is clicked the component can navigate to itself with a new “showMore” value

The property can be bound to the view thus it will be updated for each change

```
get showMore() {  
  const showMore = this.activatedRoute.snapshot.params.showMore;  
  return showMore === "true" || showMore === true;  
}  
  
async toggleMore() {  
  this.router.navigate(  
    [  
      {showMore: !this.showMore}  
    ],  
    {  
      relativeTo: this.activatedRoute  
    })  
}
```

Navigate to “self” and change the showMore settings



# Persist Preferences

17

- Using URL to store preferences means the user can refresh the browser and keep current selection
- The state can be extracted during `ngOnInit`

```
ngOnInit() {  
  this.contact = this.contactService.getById(this.id);  
}  
  
get id() {  
  return this.activatedRoute.snapshot.params.id;  
}
```

- But `ngOnInit` is executed only once during component lifetime
- How can we react to changes inside `params.id` ?

# Reactive Route Parameters

18

```

constructor(private activatedRoute: ActivatedRoute, private router: Router, private contactService: ContactService) {
  this.id = this.activatedRoute.params.map(p => p.id);
  this.contact = this.id.map(id => contactService.getById(id));
  this.name = this.contact.map(c => c.name);
  this.showMore = this.activatedRoute.params.map(p => p.showMore);
  this.showMoreCaption = this.showMore.map(more => (more ? "Less" : "More"));
}

```

Clicking the link causes a change inside `activatedRoute.params` and then all other observables are updated too

```

<h1>Contact Details {{name | async}}</h1>
<a [routerLink]="['/contact', 2]">Goto Parent</a>
<button (click)="toggleMore()">{{showMoreCaption | async}}</button>

```

The `async` pipe tells Angular to subscribe to the observable

# Nested Route

19

- A route may have children

```
<h1>Contacts</h1>

<div class="content">
  <div class="side-bar">
    <a routerLink="/all">All</a>
    <a routerLink="/top10">Top 10</a>
    <a routerLink="recent">Recent</a>
  </div>

  <div class="placeholder">
    <router-outlet></router-outlet>
  </div>
</div>
```

```
{
  path: 'contact',
  component: ContactsPageComponent,
  children: [
    {
      path: ':type',
      component: ContactListComponent,
    },
  ]
}
```

- Each route can have its own required and optional parameters

This is a relative path too !!!

contactsPage.component.html

An outlet which contains child component

# Nested routes - Imperative navigation

20

- When navigating from code relative path are resolved as absolute
- Use the **relativeTo** option to fix that

```
up() {  
  this.router.navigate([".."], {relativeTo: this.activatedRoute});  
}
```

Without relativeTo option the ".." will be resolved against the root path

Router instance is the same for all components

# Named Outlet

21

- A component may contain multiple outlets
- Each outlet must have unique name
- The names are used when navigating

```
<h1>About</h1>
```

```
<router-outlet name="left"></router-outlet>
```

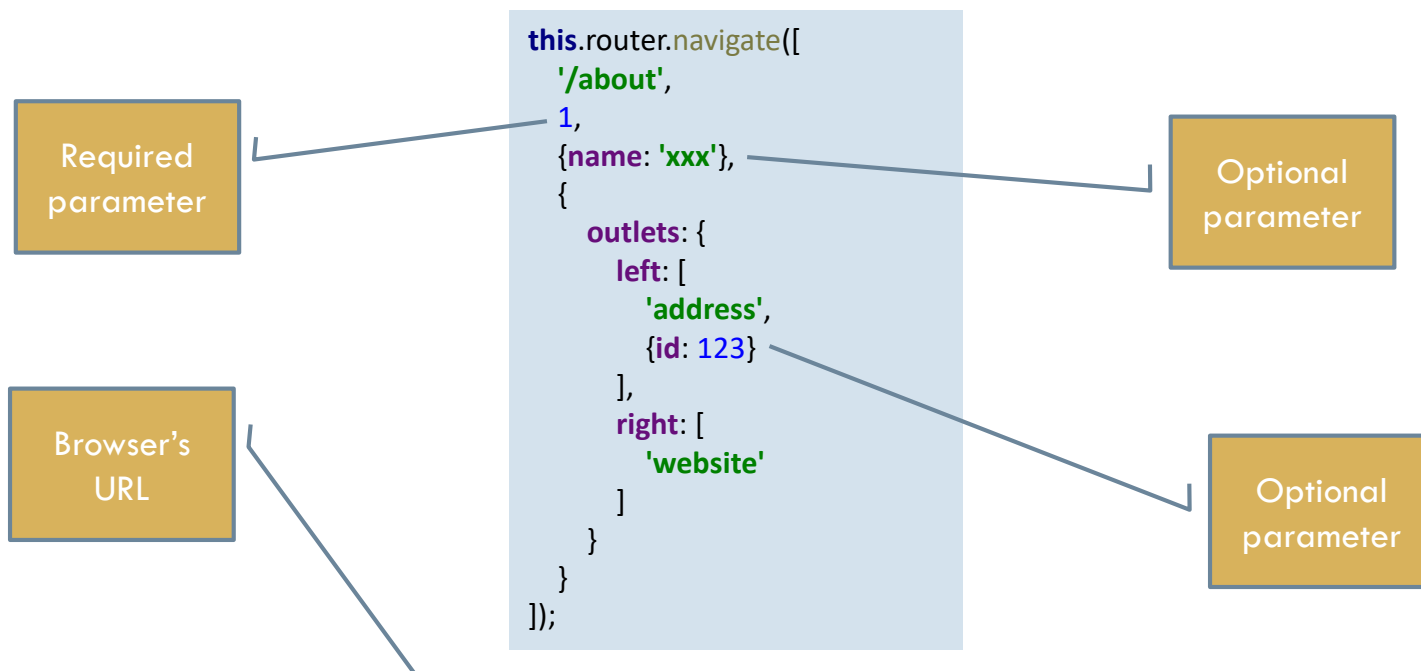
```
<router-outlet name="right"></router-outlet>
```

# Named Outlet

22

- routerLink/navigate are now much more complex since you need to specify the content for each outlet

```
<a [routerLink]="['/about', 1, {name: 'xxx'}, {outlets: {left: ['address', {id: 123}], right: ['website']}}]">About</a>
```



```
http://localhost:8080/about/1;name=xxx/(left:address;id=123//right:website)
```

# Component Lifetime

23

- By default the router dispose the component of the previous route
- This is extremely important in order to free DOM resources
- However, component's state is lost when navigating back 😞

# Sticky Routes

24

- Tell Angular to keep old route's component and reuse it when reactivating the route

```
export class CustomReuseStrategy implements RouteReuseStrategy {
  handlers: {[key: string]: DetachedRouteHandle} = {};

  shouldDetach(route: ActivatedRouteSnapshot): boolean {
    return true;
  }

  store(route: ActivatedRouteSnapshot, handle: DetachedRouteHandle): void {
    this.handlers[(<any>route).routeConfig.path] = handle;
  }

  shouldAttach(route: ActivatedRouteSnapshot): boolean {
    return !!route.routeConfig && !!this.handlers[(<any>route).routeConfig.path];
  }

  retrieve(route: ActivatedRouteSnapshot): DetachedRouteHandle | null {
    return this.handlers[(<any>route).routeConfig.path];
  }

  shouldReuseRoute(future: ActivatedRouteSnapshot, curr: ActivatedRouteSnapshot): boolean {
    return future.routeConfig === curr.routeConfig;
  }
}
```

Must register  
the custom  
strategy

```
providers: [
  {
    provide: RouteReuseStrategy,
    useClass: CustomReuseStrategy
  }
]
```



# Lazy Loading

25

- The router is capable of loading a module before navigating to one of its component

```
{
  path: 'admin',
  loadChildren: "app/admin/admin.module#AdminModule"
}
```

```
<a [routerLink]="['/admin']">Admin</a>
```

```
export const routes: Routes = [
  {
    path: "",
    component: AdminHomeComponent,
    pathMatch: 'full'
  },
];
```

```
@NgModule({
  imports: [
    CommonModule,
    RouterModule.forChild(routes)
  ],
  providers: [
  ],
  declarations: [
    AdminHomeComponent,
  ]
})
export class AdminModule {
}
```

# Lazy Loading

26

- @angular/core defines a token named **NgModuleFactoryLoader**
- @angular/router register **SystemJsNgModuleLoader** as the implementation

Uses SystemJS  
to load the file  
from the server

```
SystemJsNgModuleLoader.prototype.loadAndCompile = function (path) {  
  var _this = this;  
  var _a = path.split(_SEPARATOR), module = _a[0], exportName = _a[1];  
  if (exportName === undefined) {  
    exportName = 'default';  
  }  
  return System.import(module)  
    .then(function (module) { return module[exportName]; })  
    .then(function (type) { return checkNotEmpty(type, module, exportName); })  
    .then(function (type) { return _this._compiler.compileModuleAsync(type); });  
};
```

The module is  
expected to export  
an NgModule under  
the correct name  
(AdminModule)

# Lazy Loading

27

- But @angular/cli uses **Webpack**
- Loading Webpack chunk with SystemJS is problematic
- @angular/cli provides special Webpack plugin named **AotPlugin**
  - ▣ Parses the AST syntax looking for **loadChildren**
  - ▣ Creates chunk for every module
  - ▣ Fixes Webpack's asyncContext to allow async loading of the chunk based on the module name

# Lazy Loading - Webpack

28

SystemJS is removed

```
SystemJsNgModuleLoader.prototype.loadAndCompile = function (path) {
  var _this = this;
  var _a = path.split(_SEPARATOR), module = _a[0], exportName = _a[1];
  if (exportName === undefined) {
    exportName = 'default';
  }
  return __webpack_require__(85)(module)
    .then(function (module) { return module[exportName]; })
    .then(function (type) { return checkNotEmpty(type, module, exportName); })
    .then(function (type) { return _this._compiler.compileModuleAsync(type); });
};
```

Our main bundle now contains a map of all lazy loaded modules

```
var map = {
  "./admin/admin.module": [
    190,
    0
  ]
};

function webpackAsyncContext(req) {
  var ids = map[req]; if(!ids)
    return Promise.reject(new Error("Cannot find module '" + req + "'"));
  return __webpack_require__.e(ids[1]).then(function() {
    return __webpack_require__(ids[0]);
  });
};
```

# route.data

29

- Use it to store static metadata for a route
- Later the metadata can be used to implement application aspects like Authorization

```
{  
  path: 'admin',  
  component: AdminComponent,  
  data: {  
    roles: ["admin"],  
  },  
  canActivate: [CanActivateAdmin],  
}
```

```
export class AdminComponent {  
  constructor(activatedRoute: ActivatedRoute) {  
    console.log(activatedRoute.snapshot.data.roles);  
  }  
}
```

Set and use

# Implementing Aspects

30

- An aspect is usually marinated outside of the component → Use **route guard**

```
@Injectable()
export class CanActivateAdmin implements CanActivate {
  constructor(private authService: AuthService) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    return this.authService.isInRole(route.data.roles);
  }
}
```

```
{
  path: 'admin',
  component: AdminComponent,
  data: {
    roles: ["admin"],
  },
  canActivate: [CanActivateAdmin],
}
```

Set and use

# Associating multiple routes with the same guard

31

- Use component-less route to group multiple routes with the same configuration

Component  
-less route

Is executed  
for every  
descendent

```
export const routes: Routes = [
  {
    path: "",
    canActivate: [AuthorizeGuard],
    canActivateChild: [AuthorizeGuard],
    children: [
      {
        path: "", redirectTo: "home", pathMatch: "full",
      },
      {
        path: "home", component: HomeComponent, data: {roles: ["user"]},
      },
      {
        path: "admin", component: AdminComponent, data: {roles: ["admin"]},
      },
    ],
  },
];
```

Is executed  
only for the  
host route

# Route Guard

32

```
@Injectable()
export class AuthorizeGuard implements CanActivate, CanActivateChild {
  constructor(private authService: AuthService) {
  }

  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    console.log("canActivate", route);

    return this.authService.isInRole(route.data.roles);
  }

  canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {
    console.log("canActivateChild", route);

    return this.authService.isInRole(route.data.roles);
  }
}
```



# Async Route Guard

33

- A route guard may return a promise
- Combined with root component-less route you can simulate async application initialization

```
export class InitAppGuard implements CanActivate {  
  canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot) {  
    return new Promise(function(resolve, reject) {  
      setTimeout(function() {  
        resolve(true);  
      }, 3000);  
    });  
  }  
}
```

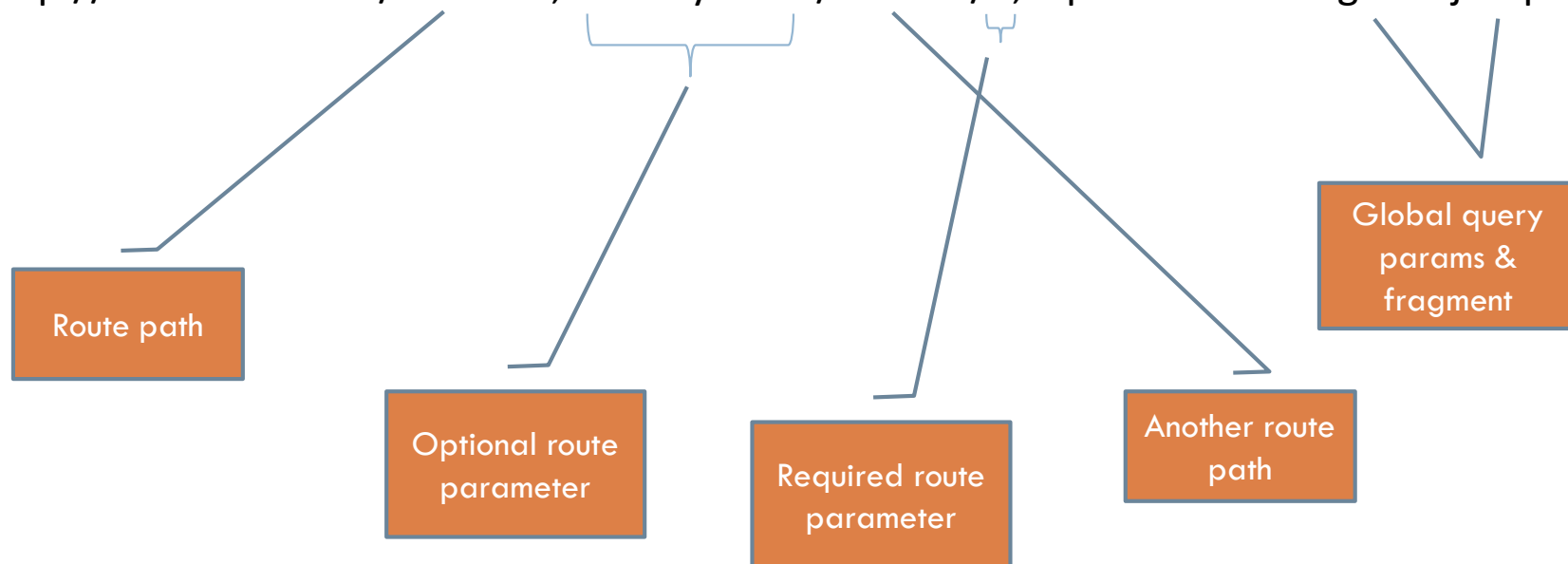
```
export const routes: Routes = [  
  {  
    path: "",  
    canActivate: [InitAppGuard],  
    children: [  
    ],  
  },  
];
```

# routerLink Parameters

34

- A link may specify additional parameters

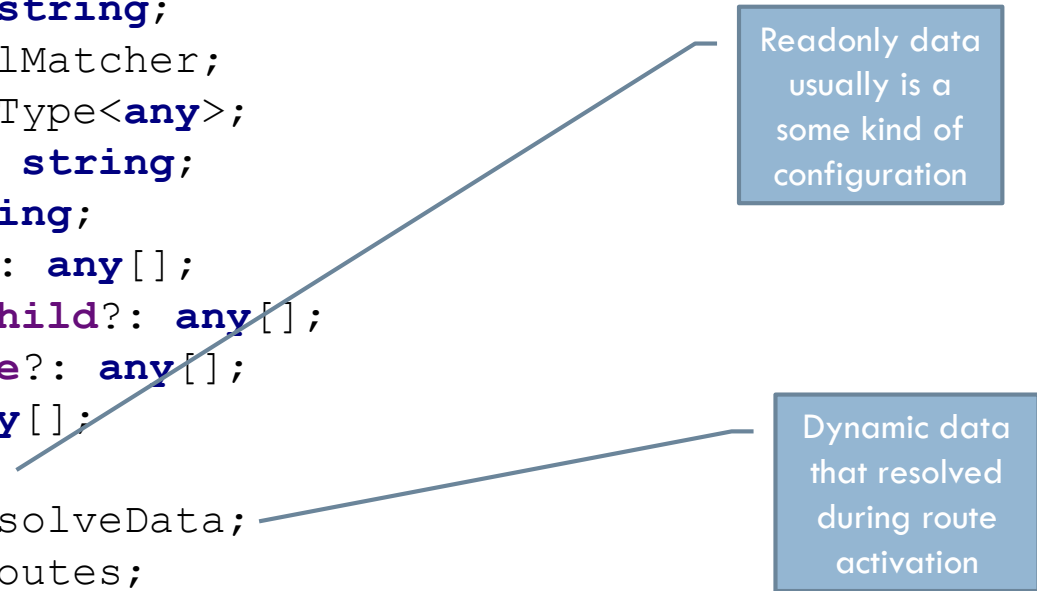
`http://localhost:8080/contacts;orderBy=desc/contact/2;expand=true?lang=en#jump`



# Route

35

```
export interface Route {  
  path?: string;  
  pathMatch?: string;  
  matcher?: UrlMatcher;  
  component?: Type<any>;  
  redirectTo?: string;  
  outlet?: string;  
  canActivate?: any[];  
  canActivateChild?: any[];  
  canDeactivate?: any[];  
  canLoad?: any[];  
  data?: Data;  
  resolve?: ResolveData;  
  children?: Routes;  
  loadChildren?: LoadChildren;  
  runGuardsAndResolvers?: RunGuardsAndResolvers;  
}
```



Readonly data  
usually is a  
some kind of  
configuration

Dynamic data  
that resolved  
during route  
activation

# Summary