incentro

Angular Advanced Module – Advanced Routing

Peter Kassenaar –
info@kassenaar.com

Contents

Named Router Outlets

Displaying multiple components in one view and updating the route

Named router outlets

- Multiple `<router-outlet>`'s in one component.
- Project different components in named outlets
  - `<router-outlet` **`name="list"`**`>…</router-outlet>`
  - `<router-outlet` **`name="detail"`**`>…</router-outlet>`
- State is reflected in URL
- Also known as *auxilliary routes*

OutletProject

localhost:4200/cities/(detail:1//list:cityList)

URL

Home Cities

List

- Groningen
- Hengelo
- Den Haag
- Enschede
- Heerlen

Details - Groningen

1

Name: Groningen

Province: Groningen

Highlights: Martinitoren

Outlets

../140-named-router-outlets

Preparation – set up routing module

- We use a simple setup:
  - One route pointing to the home route, `HomeComponent`
  - One route pointing to cities route, `CityComponent`
- CityComponent has two child routes
  - `CityListComponent`
  - `CityDetailComponent`

```
const routes: Routes = [
   {path: '', redirectTo: 'home', pathMatch: 'full'},
   {path: 'home', component: HomeComponent},
   {
      path    : 'cities', component: CityComponent,
      children: [
         {path: 'cityList', component: CityListComponent, outlet: 'list'},
         {path: ':id', component: CityDetailComponent, outlet: 'detail'}
      ]
   }
];
```

Set up router outlets

app.component.html – 1 unnamed <router-outlet>

```
<div class="container">
   <!--Main menu-->
   …
  <router-outlet></router-outlet>
</div>
```

city.component.html – 2 named <router-outlet>'s

```
<div class="row">
   <div class="col-md-6">
      <router-outlet name="list"></router-outlet>
   </div>
   <div class="col-md-4">
      <router-outlet name="detail"></router-outlet>
   </div>
</div>
```

Router navigate

Home button – static route (`app.component.html`)

```
<a class="btn btn-primary" routerLink="home">Home</a>
```

Cities button – dynamic route. Pass in the `outlets` object, and map the `list` outlet to `cityList` component and `detail` to `-1` for the moment.

```
<a class="btn btn-primary"
   [routerLink]="['/cities',
         {outlets: {'list': ['cityList'], 'detail': ['-1']}}]">
   Cities
</a>
```

Display CityList

- Default way of displaying data
  - Fetch cities via `cityService`.
  - We use the `async` pipe here

```
// city-list.component.ts
constructor(private router: Router,
         private cityService: CityService) {    }

ngOnInit(){
   this.cities=this.cityService.getCities();
}
```

```
<ul class="list-group">
   <li class="list-group-item"
      [class.selected]="city === currentCity"
      (click)="selectCity(city)"
      *ngFor="let city of cities | async">
      {{ city.name }}
   </li>
</ul>
```

Navigate to detail route

- Use `router` to show detail in the outlet:

```
selectCity(city: City) {
   console.log('navigate to: ', city.name);
   this.currentCity = city;
   this.router.navigate(['/cities', {outlets: {'detail': [city.id]}}])
}
```

Set up detail outlet using RouteParams

```
constructor(private route: ActivatedRoute,
         private  cityService: CityService) {
}

ngOnInit() {
   this.route.params.subscribe((params: { id: string }) => {
      this.id = +params.id;
      if (this.id === -1) {
         // Default: navigate to first city (or leave empty)
         this.id = 1;
      }
      this.city = this.cityService.getCity(this.id);
   })

}
```

Displaying the detail data

For your reference. This is easy:

```
<div *ngIf="city | async; else loading; let city">
   <ul class="list-group">
      <li class="list-group-item">Name: {{ city.name }}</li>
      <li class="list-group-item">Province: {{ city.province }}</li>
      <li class="list-group-item">Highlights: {{ city.highlights }}</li>
   </ul>
   <img src="assets/img/{{ city.name}}.jpg" alt="Foto van {{ city.name }}" />
</div>
<ng-template #loading>
   <div>
      <h3>Fetching city...</h3>
      <img src="assets/img/spinner.gif" alt="loading...">
   </div>
</ng-template>
```

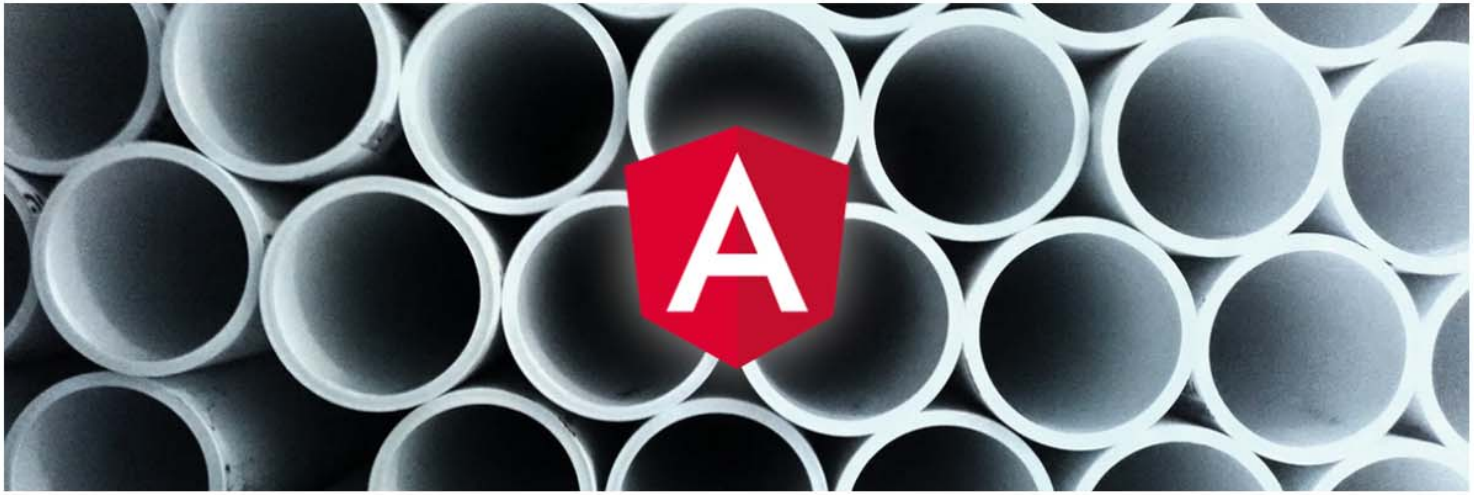
More info on named router outlets

ONEHUNGRYMIND

ABOUT

Dreams Do Come True! Named Router Outlets in Angular 2

Lukas Ruebbelke



Intro

I am generally a positive person that endeavors to say nice things about people and frameworks. In the broadest sense, I love Angular, and it has been an amazing tool to build some really cool things. Angular 2 has exceeded my expectations in a lot of ways.

http://onehungrymind.com/named-router-outlets-in-angular-2/

RANGLE.IO SERVICES PROCESS WORK EVENTS RESOURCES ABOUT BLOG INQUIRE

Type to search

master

Using Auxiliary Routes

Angular supports the concept of auxiliary routes, which allow you to set up and navigate multiple independent routes in a single app. Each component has one primary route and zero or more auxiliary outlets. Auxiliary outlets must have unique name within a component.

To define the auxiliary route we must first add a named router outlet where contents for the auxiliary route are to be rendered.

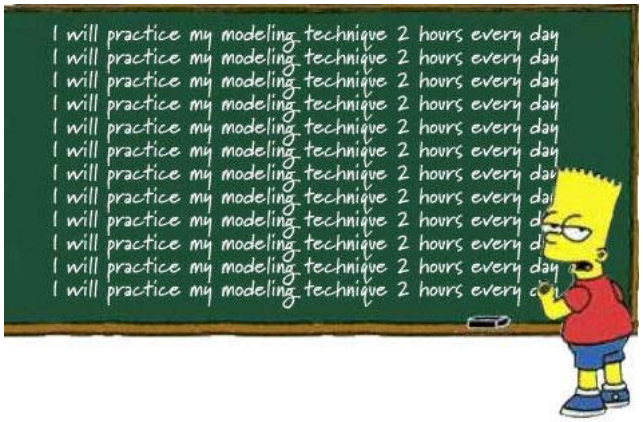
Here's an example:

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

@Component({
  selector: 'app',
  template: `
    <nav>
      <a [routerLink]="['/component-one']">Component One</a>
      <a [routerLink]="['/component-two']">Component Two</a>
      <a [routerLink]="[{ outlets: { 'sidebar': ['component-aux'] } }]">Component Aux</a>
    </nav>

    <div style="color: green; margin-top: 1rem;">Outlet:</div>
```

https://angular-2-training-book.rangle.io/handout/routing/aux-routes.html

Workshop

- Open `../140-named-router-outlets` and work from there
- OR: start a new project using routing and set it up using the previous slides.
- Add a new, named router outlet to `city.component.html`
- Create a new component to be shown in this outlet
- Create a link, navigating to this new outlet, showing the component
  - Place the link inside `city-list.component.html`.
  - Look at the `[routerLink]` notation!



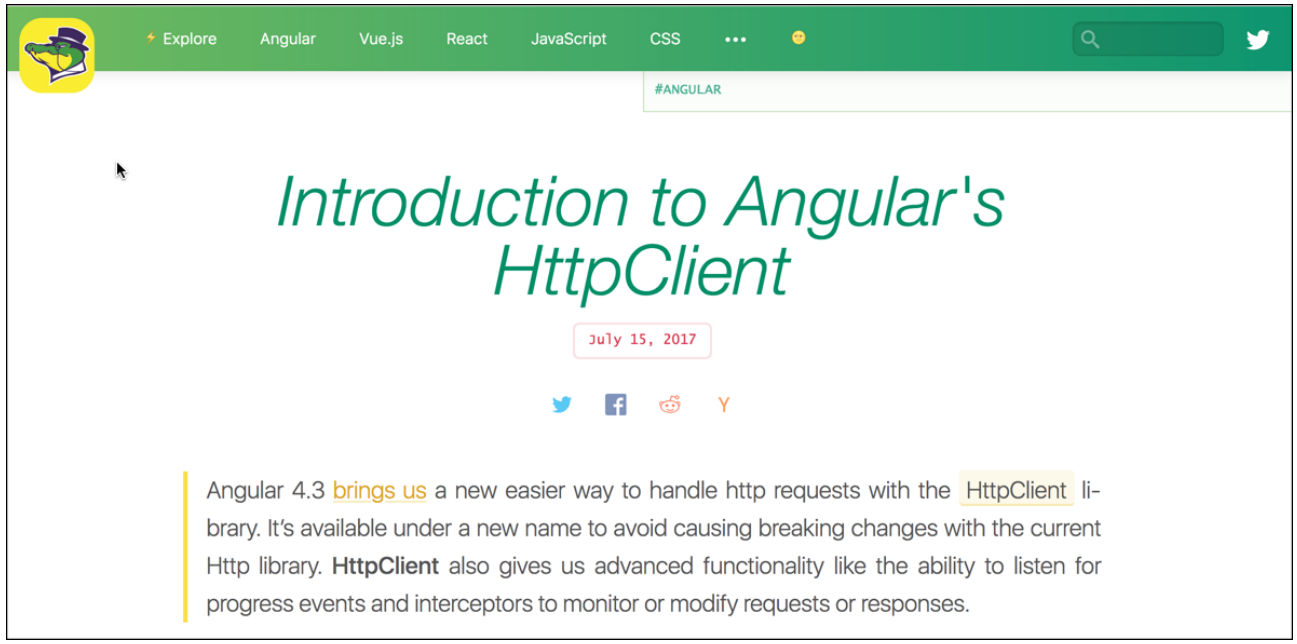
HttpClientModule

On Angular 4.3+ projects

Angular 4.3+: HttpClientModule

- In @ngModule: `imports : [HttpClientModule]`
- No more `.map(res => res.json())`.
  - Json is default!
  - Add params if you want access to raw Response Object
- New: Interceptors
- https://alligator.io/angular/httpclient-intro/ and
- https://alligator.io/angular/httpclient-interceptors/

- ...is default in Angular 5
  - `HttpModule` will be removed in future versions
  - Update your Angular-CLI!
  - `HttpModule` (Angular 4) vs.
  - `HttpClientModule` (Angular 4.3+)



https://alligator.io/angular/httpclient-intro/

Routing events are actually Observables.

Which means we can subscribe! And do something like:

```
this.router.events
    .subscribe(event =>{
      console.log(' router event: ', event);
  })
```

(Don’t forget to inject router:

```
constructor(private router: Router) {  })
```

Or, in a more reactive way of programming:

```
this.router.events
    .filter(event => event instanceof NavigationEnd)
    .map(...)
    . ...
    .subscribe(event =>{
      console.log(' router event: ', event);
  })
```

RxJS 5.5+: lettable operators en .pipe()

This repository Search Pull requests Issues Marketplace Explore

ReactiveX / rxjs

Watch 357 Star 9,548 Fork 905

Code Issues 225 Pull requests 68 Projects 3 Insights

Branch: master rxjs / doc / lettable-operators.md Find file Copy path

seniorquico docs: change "pipe" utility function module path from "utils" to "uti... 39c2e7c on Oct 25

6 contributors

237 lines (187 sloc) 9.12 KB Raw Blame History

Lettable Operators

Starting in version 5.5 we have shipped "lettable operators", which can be accessed in `rxjs/operators` (notice the pluralized "operators"). These are meant to be a better approach for pulling in just the operators you need than the "patch" operators found in `rxjs/add/operator/*`.

NOTE: Using `rxjs/operators` without making changes to your build process can result in larger bundles. See Known Issues section below.

Renamed Operators

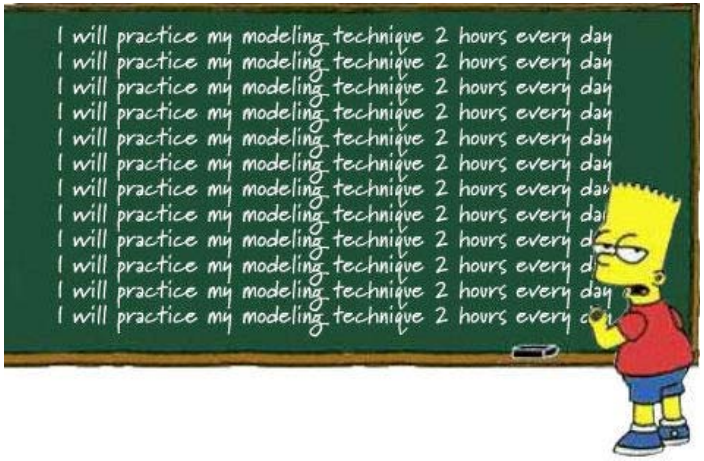
Due to having operators available independent of an Observable, operator names cannot conflict with JavaScript keyword restrictions. Therefore the names of the lettable version of some operators have changed. These operators are:

https://github.com/ReactiveX/rxjs/blob/master/doc/lettable-operators.md

```
// new: rxjs 5.5 lettable operators with .pipe()
return
this.http.get<City[]>('assets/data/cities.json')
  .pipe(
    tap(res => console.log(res)),
    catchError(err => {
      console.log(err);
      return Observable.of([])
    })
  )
```

Workshop

- Use the example `../140-named-router-outlets`
- Replace old skool `HttpModule` with `HttpClientModule`:
  - Update `app.module.ts`
  - Update `city.service.ts`
- Read post on `HttpClientModule` by Alligator
  - Optional: create an `Interceptor` – Use it simply to log current Time and the requested page to the console.



Router Guards

Securing parts of your route

Guard Types

- Four types of guards:
  - `CanAcativate` – decides if a route can be activated
  - `CanActivateChild` – decides if children of a route can be activated
  - `CanDeactivate` – decides if a route can be deactivated
  - `CanLoad` – decides if a module can be loaded lazily

Credits: http://blog.thoughtram.io/angular/2016/07/18/guards-in-angular-2.html

Defining Guards

- Multiple ways (as functions or as classes)
- Regardless, it needs to return a
  - `Observable<boolean>`,
  - `Promise<boolean>` or
  - `boolean`.
- Defined in `@NgModule`, or as a separate class

Guards as a function

- Define a token and a guard function. For example in `app.module.ts`.

```
@NgModule({
  …
  providers: [
    CityService,
    AuthService,
    {
      provide: 'CanAlwaysActivateGuard', // Guard as a function
      useValue: guardFunction
    },
    CanActivateViaAuthGuard,
    CanDeactivateGuard
  ],
  bootstrap: [MainComponent]
})
export class AppModule {}
```

Token

Function

```
export function guardFunction() {
  console.log('Route requested');
  return true; // do validation or other stuff here
}
```

Use the guard token in app.routes

```
// app.routes.ts
…
export const AppRoutes: Routes = [
    …
    {
        path: 'home',
        component: AppComponent,
        canActivate: ['CanAlwaysActivateGuard'] // Defined in app.module.ts
    },
    …
];
```

(re)use of string token

You *can* have multiple tokens/functions, guarding your route

Guards as a class – *most used*

- Used: when the guard needs Dependency Injection
- Common use: with some kind of Authentication Service.

- All about Implementing interfaces!
  - `canActivate()`
  - `canActivateChild()`
  - `canDeActivate()`

canActivateViaAuthGuard.ts

```
// canActivateViaAuthGuard.ts
import { Injectable } from '@angular/core';
import { CanActivate } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable()
export class CanActivateViaAuthGuard implements CanActivate {

   constructor(private authService: AuthService) {}

   canActivate() {
      return this.authService.isLoggedIn();
   }
}
```

Class/Guard name

Auth Service

Interface implementation

Register Guard class on module and routes

```
// app.module.ts
…
@NgModule({
   …
   providers   : [
      …,
      AuthService,
      CanActivateViaAuthGuard
   ],
   …
})
export class AppModule {
}
```

```
// app.routes.ts
…
import {CanActivateViaAuthGuard} from "./canActivateViaAuthGuard";

export const AppRoutes: Routes = [
   …
   {
      path        : 'add',
      component   : CityAddComponent,
      canActivate: [CanActivateViaAuthGuard]
   },
   …
];
```

Deactivating routes

- Called when navigating *away* from a route
- Same approach as `CanActivate` route

```
// canDeactivateGuard.ts
import {Injectable} from '@angular/core';
import {CanDeactivate} from '@angular/router';
import {CanDeactivateComponent} from "./canDeactivate.component";

@Injectable()
export class CanDeactivateGuard implements CanDeactivate<CanDeactivateComponent> {

   canDeactivate(target:CanDeactivateComponent) {
      // Can the user deactivate the route? Test for changes here!
      // For now, return Yes|Nope from the browser confirm dialog.
      if (target.hasChanges()) {
         return window.confirm('Do you really want to cancel? There might be unsaved changes
      }
      return true;
   }
}
```

Add guard to routes

```
// app.routes.ts
…
import {CanDeactivateComponent} from "./canDeactivate.component";
import {CanDeactivateGuard} from "./canDeactivateGuard";

export const AppRoutes: Routes = [
   …
   {
      path          : 'deactivate',
      component     : CanDeactivateComponent,
      canDeactivate: [CanDeactivateGuard]
   },
   ….
];
```

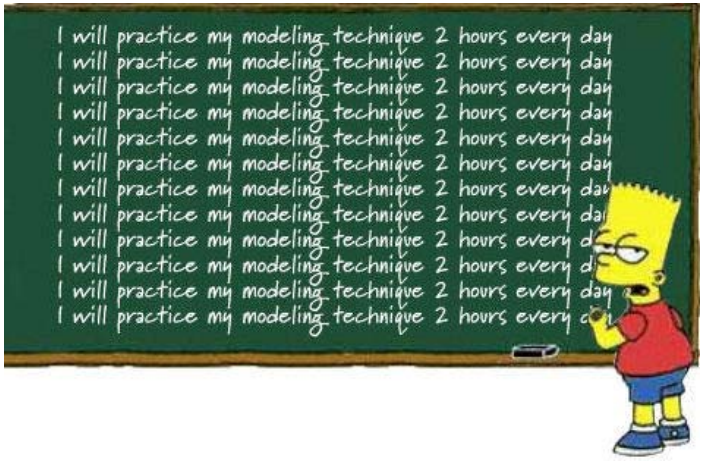
Create DeactivateComponent

- Add implementation of .hasChanges()!

```
// …
export class CanDeactivateComponent implements OnInit {
   // Properties voor de component/class
   myForm:FormGroup = new FormGroup({
      txtInput:new FormControl()
   });

   constructor(private route: Router) { }
   ngOnInit() {}

   moveAway() {
      this.route.navigate(['/home']);
   }
   hasChanges(){
      return this.myForm.dirty; // return state of the form
   }
}
```

Workshop

- Use the example `../150-router-guards`
- TBD..



Route Resolvers

Fetching data *before* a component is loaded

What are route resolvers

- Default way of doing things
  - Navigate to a specific route/component
  - Show a loading indicator of some kind
  - Fetch data inside the component in `ngOnInit()` or `constructor()`.

- With a route resolver
  - First fetch the data
  - Then show the component
  - No need to add the safe navigation operator (aka Elvis operator) like `{{ user?.firstname }}`

How do resolvers work – SimpleResolver

Step 1 - Create a separate class (or rather: a Service) for the resolver

```
// app.resolver.service.ts
import {Injectable} from '@angular/core';
import {Resolve} from '@angular/router';
…
@Injectable({
  providedIn: 'root'
})
export class SimpleResolverService implements Resolve<Observable<string>> {

  // Creating a resolver is all about implementing
  // the Resolve interface with a specific type
  resolve() {
    return of('Hello World').pipe(
      delay(2000)
    );
  }
}
```

Interface

Simple implementation

Step 2 – set up route configuration

Add the `resolve` key to the configuration and call the provide service.

Here, the resolved data will be available under the `message` key.

```
const routes: Routes = [
  …,
  {
    path: 'simple',
    component: SimpleResolverComponent,
    resolve: { message: SimpleResolverService}
  }

];
```

Step 3 – use resolved data in the component

Use the `data` property on the ActivatedRoute's `snapshot` object

The component is showed after a 2 sec. delay.

```
…
@Component({
  selector: 'app-simple-resolver',
  template: `
   …
    {{ data.message }}
  `
})
export class SimpleResolverComponent implements OnInit {

  data: any;
  constructor(private  route: ActivatedRoute) { }

  ngOnInit() {
    this.data = this.route.snapshot.data;
  }

}
```

Inject and use `ActivatedRoute`

Resolving real data from an API

Same process. Only talk to an external API

```
// app.resolver.service.ts
import {Injectable} from '@angular/core';
import {Resolve} from '@angular/router';
import {Observable} from 'rxjs';
import {HttpClient} from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class ApiResolverService implements Resolve<Observable<any>> {

  // Talk to external API
  url = 'https://randomuser.me/api/?results=10';

  constructor(private  http: HttpClient) {
  }

  resolve() {
    return this.http.get<any>(this.url);
  }

}
```

Inject and use `HttpClient`

2. Update routing

```
import {ApiResolverComponent} from './api-resolver/api-resolver.component';
…
const routes: Routes = [
  …
  {
    path: 'api',
    component: ApiResolverComponent,
    resolve: {data: ApiResolverService}
  }
];
…
export class AppRoutingModule {
}
```

3. Use the provided data

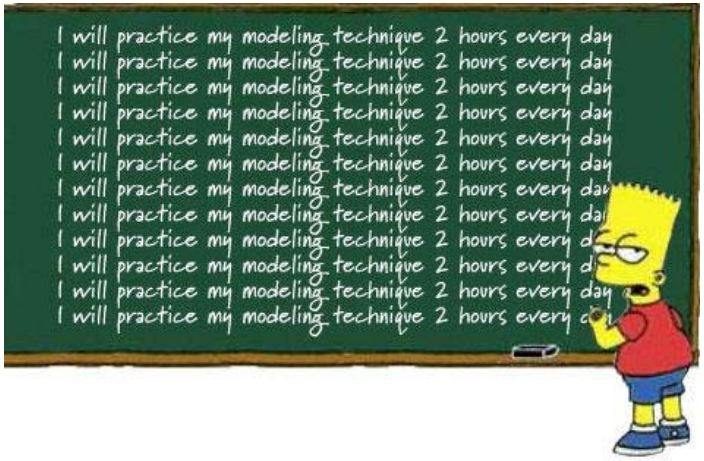
- You *know* it's available in the component, b/c of the resolver
- No need for async binding or using `{{ user?.name }}`
- Inspect and optionally model the `data` you get back from the API (recommended)

```
<ul class="list-group">
  <li *ngFor="let user of data" class="list-group-item">
    <h4>{{ user.name.title}} {{ user.name.first }} {{ user.name.last}}</h4>
    ({{ user.email}})
  </li>
</ul>
```

```
export class ApiResolverComponent implements OnInit {
  data: any;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    // inspect the object that the API returns!
    this.data = this.route.snapshot.data.data.results;
  }
}
```

Workshop

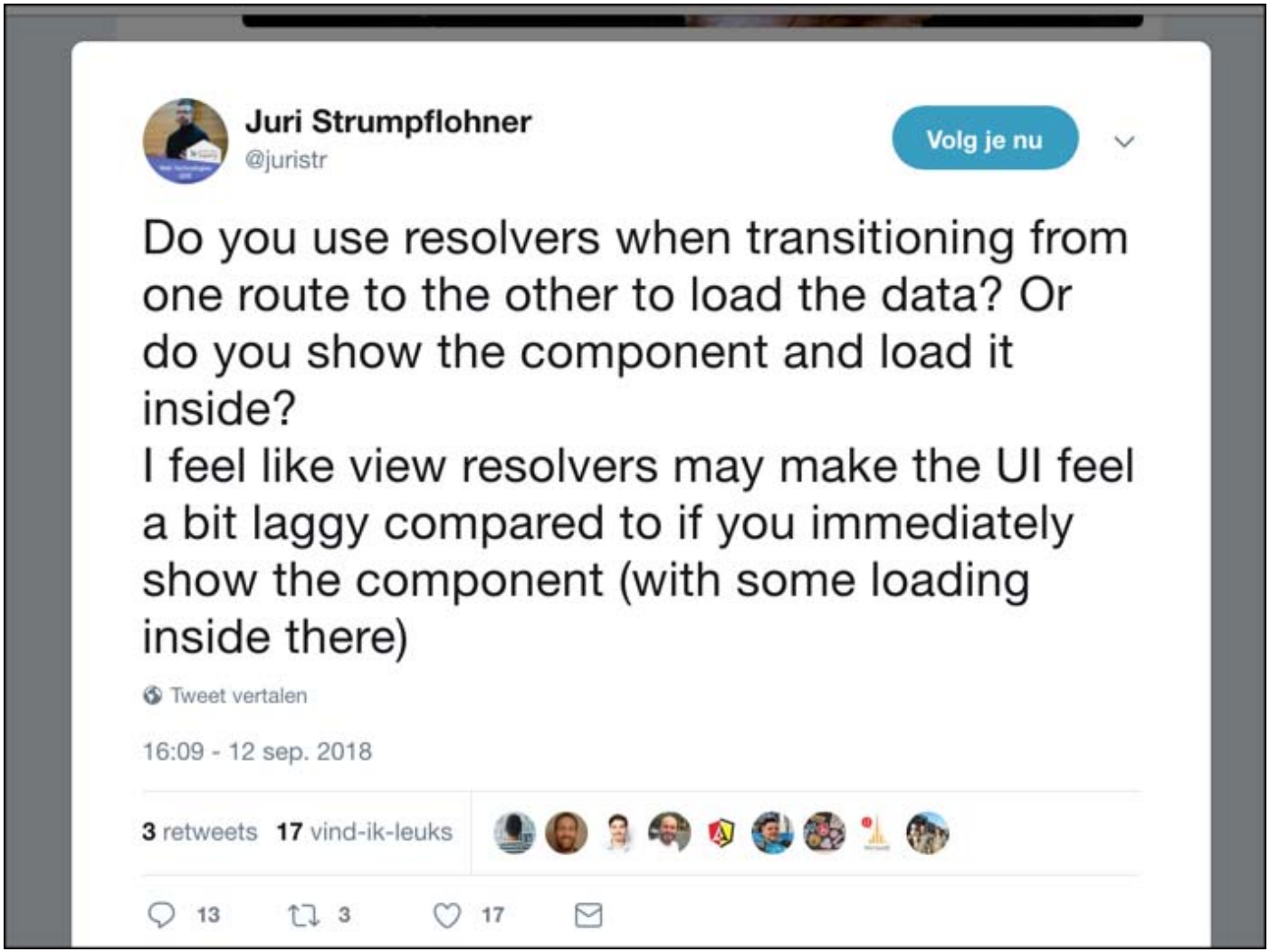
- Use the example `../160-route-resolvers`
- TBD..



Resolvers - yep or nope?

- Discussion – it might not give the best possible UX
- Yep – if you *don't* use resolver and data fails to load, you're stuck on the 'wrong' view.
- Nope – Adds lag to application, just load component w/ spinner to show something is happening.
  - Application can seem to freeze while waiting for the response.

- Read the replies/discussion on https://twitter.com/juristr/status/1039878711468810245



- https://stackoverflow.com/questions/49054232/why-would-you-use-a-resolver-with-angular

stackoverflow NEW Search...

Home

PUBLIC

Stack Overflow

Tags

Users

Jobs

Teams
Q&A for work

Learn More

Why would you use a resolver with Angular

13

4

I like the idea of `resolver`s.

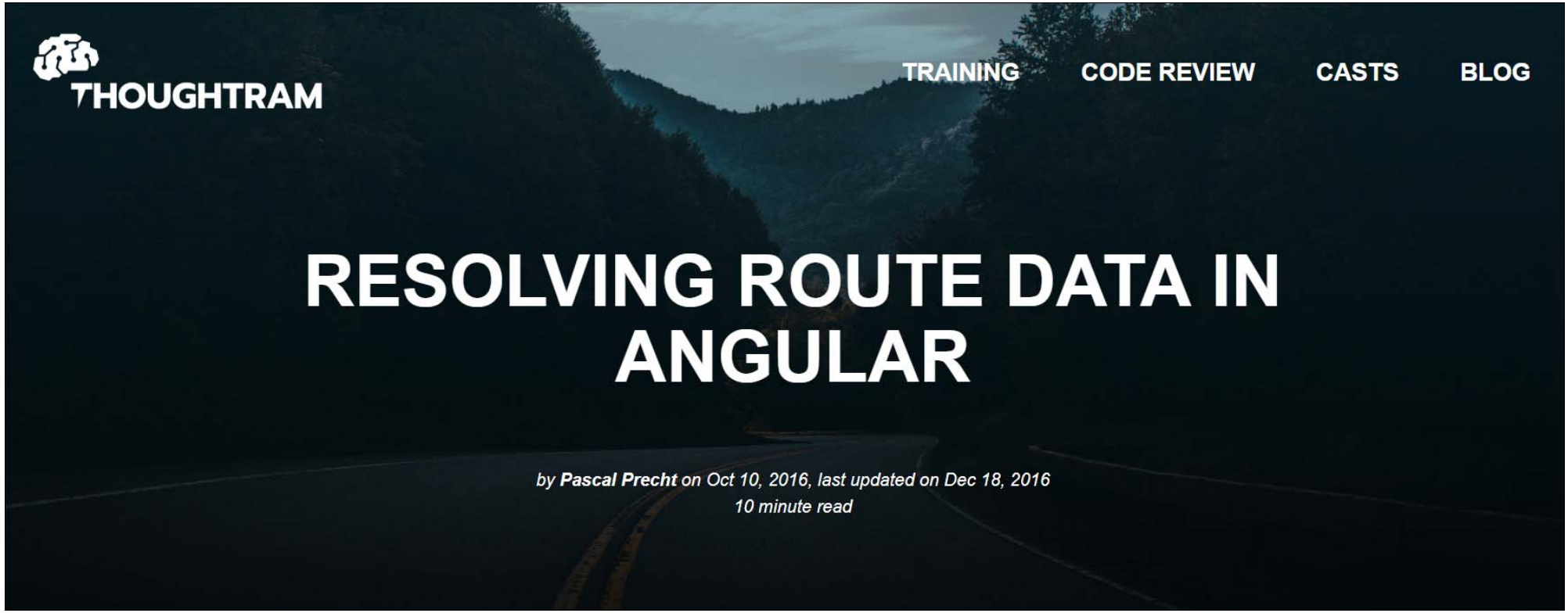
You can say that:
- for a given route you expect some data to be loaded first
- you can just have a really simple component with no observable (as retrieving data from `this.route.snapshot.data`)

So resolvers makes a lot of sense.

BUT:
- You're not changing the URL and displaying your requested component until you receive the actual response. So you can't (simply) show the user that something is happening by rendering your component and showing as much as you can (just like it's advised to, for the shell app with PWA). Which means that when having a bad connection, your user may have to just wait without visual indication of what's happening for a long time

More info on Resolvers

- https://blog.thoughtram.io/angular/2016/10/10/resolving-route-data-in-angular-2.html



More info on resolvers

- https://angular.io/guide/router#resolve-guard
- https://alligator.io/angular/route-resolvers/
- https://codeburst.io/understanding-resolvers-in-angular-736e9db71267