

# ANGULAR 9



Peter Gurský, [peter.gursky@upjs.sk](mailto:peter.gursky@upjs.sk)

# Téma: komunikácia komponentov

- **Ciel'**: Vytvoríme si nezávislý komponent s formulárom, v ktorom budeme môcť editovať existujúcu alebo novú skupinu
- V `src/modules/groups` spustíme
  - ▣ `ng g c group-edit-child`
- Ak ho chceme vidieť v komponente `GroupDetailComponent`, vložíme do jeho šablóny tag, ktorý sa volá rovnako, ako `selector` v novom komponente a importujeme ho

```
<app-group-edit-child></app-group-edit-child>
```

# group-edit-child pre pridávanie aj editáciu

- Potrebujeme, aby rodičovský komponent informoval o tom, koho editujeme
- V `group-edit-child.component.ts` si zadefinujeme, ktoré inštančné premenné chce mať na vstupe
  - ▣ Rodičovský komponent nevolá konštruktor, to robí framework
  - ▣ posielame premennú `group` (nová alebo editovaná),
- V rodičovi zadefinujeme hodnoty pre tieto premenné

# Odoslanie vstupu od rodiča

## group-edit-child-component.ts

```
import {Input, OnChanges} from '@angular/core';  
  
export class GroupEditChildComponent implements OnChanges {  
  @Input() public group: Group;  
  ...  
  ngOnChanges() {  
    //spracovanie group premennej  
  }  
}
```

## rodič - šablóna:

```
<app-group-edit-child [group]="selectedGroup" ></app-group-edit-child>
```

inštančná premenná rodiča

# Odoslanie skupiny rodičovi

## group-edit-child-component.ts

```
import {EventEmitter, Output } from '@angular/core';

export class GroupEditChildComponent ...{
  @Output() groupChange = new EventEmitter<Group>();
  ...
  onSubmit() {
    this.groupChange.emit(this.group);
  }
}
```

## rodič - šablóna

```
<app-group-edit-child (groupChange)="onGroupSave($event)"></app-group-edit-child >
```

## rodič - komponent

```
onGroupSave(group:Group) {
  this.groups.push(group);
}
```

# Kontrola routovania

- Niekedy nechceme dovoliť navigáciu
  - používateľ nie je prihlásený
  - chceme, aby sa používateľ najprv prihlásil
  - je potrebné dodať dáta
  - neboli uložené zmeny vo formulári
  - chceme sa používateľa spýtať či danú zmenu chce uložiť
- Strážca (guard) vráti buď
  - boolean, o tom či, či navigáciu povoliť, alebo nie, alebo
  - UrlTree – sparsovaná URL, kam sa chceme presunúť
  - Observable alebo Promise z dvoch vyššie spomenutých
- Navigácia pokračuje, až keď strážca hodnotu vráti

# Typy strážcov

- Strážcovia sú funkcie
  - ▣ CanActivateFn – stráži navigáciu na konkrétne pravidlo
  - ▣ CanActivateChildFn – stráži rodiča aj deti podľa toho či sa deti môžu zobrazit'
  - ▣ CanDeactivateFn – stráži odchod z aktuálnej url
  - ▣ ResolveFn – poskytuje dáta pred aktiváciou pravidla
  - ▣ CanMatchFn – ak strážca nepustí aktuálne pravidlo, nechá router skúšať ďalšie pravidlá

# Vytvorenie strážcu

- ng generate guard guards/auth
  - vyberieme CanActivate
  - vytvorí súbor src/app/guards/auth.guard.ts

```
import { ActivatedRouteSnapshot, CanActivateFn, RouterStateSnapshot, UrlTree } from
 '@angular/router';
import { Observable } from 'rxjs';

export const authGuard: CanActivateFn = (route : ActivatedRouteSnapshot, state:
 RouterStateSnapshot)
 :Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree => {
  return true;
};
```



# Vytvorenie strážcu

objekt aktivovaného (**budúceho**) pravidla routra, ktoré strážime. Máme prístup napr. ku

- spárovanému komponentu,
- vnoreným pravidlám aj rodičovskému pravidlu,
- častiam URL, ktorá bola použitá
- dátam/parametrom, ktoré cez URL prišli

`state.url` nám povie string-ovú reprezentáciu URL, ktorú práve router spracováva

auth

guards/auth.guard.ts

```
import { ActivatedRouteSnapshot, CanActivateFn, RouterStateSnapshot, UrlTree } from '@angular/router';
import { Observable } from 'rxjs';

export const authGuard: CanActivateFn = (route : ActivatedRouteSnapshot, state: RouterStateSnapshot)
:Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree => {
  return true;
};
```

# Použitie strážcu

- sprážca stráži celý podstrom trás
  - ▣ do podtrás sa vnoríme, iba keď strážca pustí rodiča
  - ▣ ak chceme chrániť deti z rodiča použijeme `CanActivateChildFn`

```
const groupsRoutes: Routes = [  
  {  
    path: 'groups',  
    component: GroupsComponent,  
    canActivate: [authGuard],  
    children: [  
      ...  
    ]  
  }  
];
```

# Použitie strážcu

- strážca stráži celý podstrom pravidiel
  - ▣ do podpravidiel sa vnoríme, iba keď strážca pustí rodiča
  - ▣ ak chceme chrániť detské pravidlá z rodiča použijeme `CanActivateChildFn`
    - Vieme pristupovať k premenným URL určeným pre deti
    - Ak nepustí dieťa, nevytvorí ani rodiča
- poradie overovania:
  - ▣ najskôr sa overujú strážcovia `CanDeactivateFn` a `CanActivateChildFn` z najhlbšieho vnorenia pravidiel až po najvyššie,
  - ▣ potom sa overujú `CanActivateFn` strážcovia z najvyššieho pravidla po najnižšie
  - ▣ ak ktorýkoľvek strážca vráti `false/UrlTree`, ďalší strážcovia sa neoverujú

# Presmerovanie na LoginComponent

```
export const authGuard: CanActivateFn = (route, state) => {  
  const router = inject(Router);  
  const userService = inject(UsersService);  
  if (userService.isLoggedIn()) {  
    return true;  
  }  
  userService.redirectAfterLogin = state.url;  
  router.navigateByUrl('/login');  
  return false;  
};
```

alternatívne:

```
const urlTree = router.parseUrl('/login');  
return urlTree;
```

# Strážca CanDeactivateFn

- používateľ vyplnil časť formulára a uklikol sa do menu
  - ▣ používateľ a sa chceme spýtať, či chce zahodiť prácu
- používateľ poslal dáta na server, ale ešte nedošiel výsledok a už sa preklikáva inde
  - ▣ Čo ak sa uloženie nepodarilo? Používateľ o tom nebude vedieť. Môžeme počkať kým sa server vyjadrí a až potom presmerovať
- Vytvoríme si všeobecného strážcu pre komponenty, ktorý odovzdá rozhodnutie na metódu `canDeactivate()` vytvorenú v komponente

# Strážca CanDeactivate

ng g guard guards/can-deactivate

CanDeactivateFn sa viaže na konkrétny komponent, resp. v našom prípade na komponenty implementujúce interface **CanComponentDeactivate**

```
export interface CanComponentDeactivate {  
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;  
}  
  
export const canDeactivateGuard: CanDeactivateFn<CanComponentDeactivate> = (component,  
  currentRoute, currentState, nextState) => {  
  return component.canDeactivate();  
};
```

# Príklad chráneného komponentu

```
export class LoginComponent implements OnInit, CanComponentDeactivate {
  private auth: Auth = new Auth();
  constructor(private router: Router, private userService: UsersService) {}

  onSubmit() {
    this.userService.login(this.auth).subscribe(
      loginStatus => {
        if (loginStatus) {
          this.auth = new Auth();
          this.router.navigateByUrl('/');
        }
      }
    );
  }

  canDeactivate(): boolean | Observable<boolean> | Promise<boolean> {
    const canLeave = !(this.auth.name || this.auth.password);
    if (canLeave) return true;

    const confirmation = window.confirm(
      'Are you sure to leave? The form is partially filled and will be discarded.'
    );
    return of(confirmation);
  }
}
```

# Príklad chráneného komponentu

```
export class LoginComponent implements OnInit, CanComponentDeactivate {
  private auth: Auth = new Auth();
  constructor(private router: Router, private userService: UsersService) {}

  onSubmit() {
    this.userService.login(this.auth)
      .subscribe(loginStatus => {
        if (loginStatus) {
          this.auth = new Auth();
          this.router.navigateByURL('');
        }
      });
  }

  canDeactivate(): boolean | Observable<boolean> {
    const canLeave = !(this.auth.name);
    if (canLeave) return true;

    const confirmation = window.confirm(
      'Are you sure to leave? The form is partially filled and will be discarded.');
```

V pravidle zaevidujeme strážcu:

```
{
  path: 'login',
  component: LoginComponent,
  canDeactivate: [canDeactivateGuard]
}
```

```
return of(confirmation);
}
```



# ResolveFn – získanie dát pred aplikovaním pravidla

- ak chceme pred renderovaním stránky získať/pripraviť dáta, aby sa už zobrazil iba finálny komponent
  - ▣ ak neviem dodať dáta, môžem sa rovno preroutovať inde a nevytvárať komponent
- Postup:
  - ▣ vytvoríme funkciu implementujúcu **ResolveFn** na získanie dát, ktorá sa zavolá pred vytváraním komponentu v trase
    - ak vráti dáta, vezmeme si ich v komponente v `ngOnInit()` z `ActivatedRoute`
    - ak nastane chyba a teda nevráti dáta, router presmerujeme inam, napr. naspäť

# Resolver na získanie konkrétnej skupiny

Bud' v UsersService.ts:

```
public groupResolver(route: ActivatedRouteSnapshot,
                    state: RouterStateSnapshot):
                    Observable<Group> | Observable<never> {
  const strId = route.paramMap.get('id');
  return strId ? this.getGroup(parseInt(strId)) : EMPTY;
}
```

Alebo hocikde:

```
export const groupResolver: ResolveFn<Group> = (route, state) => {
  const usersService = inject(UsersService);
  const strId = route.paramMap.get('id');
  return strId ? usersService.getGroup(parseInt(strId)) : EMPTY;
}
```

# Komponent a pravidlo

```
export class GroupDetailComponent implements OnInit {
  group: Group;

  constructor(private route: ActivatedRoute) {}

  ngOnInit() {
    this.route.data.subscribe(
      (data: { group: Group }) => {
        this.group = data.group;
      });
  }
  ...
}
```

V pravidle namapujeme resolver na premennú group:

```
{
  path: ':id',
  component: GroupDetailComponent,
  resolve: {
    group: groupResolver
  }
}
```

# Strážca CanMatchFn

- použije sa v trase, kde sa robí loadChildren

```
{  
  path: 'group',  
  loadChildren: () => import('./groups/groups.module'),  
  canMatch: [authMatchGuard]  
}
```

- metódu canMatchFn píšeme takmer analogicky ako canActivateFn
  - ▣ vracia boolean | Observable<boolean> | Promise<boolean>
    - true ak chceme pravidlo aplikovať,
    - false ak nechceme – tiež môžeme pred vrátením false prenavigovať inam
  - ▣ na vstupe metódy dostaneme route:Route
    - URL získame cez route.path

# Preloading

- máme 2 možnosti
  - ▣ defaultne sa všetky moduly, ktoré majú nastavený lazy loading cez loadChildren natiahnu až po naroutovaní na príslušnú URL
  - ▣ preloadingStrategy – loaduje sa na pozadí každý modul, ktorý určí stratégia a nie je chránený s CanMatchFn
    - PreloadAllModules – stratégia vyberá každý modul
    - vlastná stratégia – implementujúca PreloadingStrategy

# nastavenie preloadingStrategy

- pridáme parameter do metódy provideRouter() hlavného konfiguračného súboru

app.config.ts

```
export const appConfig: ApplicationConfig = {
  providers: [
    provideRouter(
      routes,
      withPreloading(PreloadAllModules)
    ),
    ...
  ],
};
```

# Vlastná stratégia

- dodáme si do pravidla vlastné dáta

```
{  
  path: 'group',  
  loadChildren: () => import('./groups/groups.module'),  
  data: { preload: true }  
}
```

- vyrobíme vlastnú stratégiu, v ktorej automaticky natiahneme na pozadí iba také moduly, ktoré majú v dátach pravidla `preload == true`
- ostatné sa dotiahnu až po preroutovaní na nich
- v hlavnom routri nahradíme svoju stratégiu za `PreloadAllModules`

# Vlastná stratégia

```
import { Injectable } from '@angular/core';
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';

@Injectable({
  providedIn: 'root',
})
export class SelectivePreloadingStrategyService implements PreloadingStrategy {

  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data && route.data.preload) {
      console.log('Preloaded: ' + route.path);
      return load();
    } else {
      return of(null);
    }
  }
}
```



# Routovanie – čo sme nebrali

- animácie pri navigácii medzi pravidlami
  - ▣ <https://angular.io/guide/router#adding-routable-animations>
- paralelné routovanie v pomenovaných outletoch
  - ▣ <https://angular.io/guide/router#displaying-multiple-routes-in-named-outlets>

# Konfiguračné premenné

- Vlastnosti, ktoré chceme inak vo vývoji a inak pri nasadení chceme mať v konfiguračnom súbore
- `ng g environments`
- Pri vývoji sa použije súbor `environment.development.ts`
  - ▣ `ng serve`
- Pri vytváraní balíčka pre server sa použije súbor `environment.ts`
  - ▣ `ng build`
- Pri importovaní v kóde používame cestu k `environment.ts`

```
environment.development.ts:
```

```
export const environment = {  
  serverUrl: "http://localhost:8080/"  
};
```

# Použitie signálov

## □ Signál

- Na synchrónnu komunikáciu (nie na čakajúce operácie)
- Modelová premenná v MVC
  - Listenery sú informované hneď, ako sa hodnota v signále zmení
  - Listenerom môže byť
    - Element v DOM modeli (prekreslí sa – iba on)
    - Sledovače metód – napr. generátory ďalších signálov
      - `const doubleCount = computed(() => count() * 2);`
      - Spúšťajú stále znova vnútornú metódu, ak sa hodnoty signálov vo vnútornej metóde zmenia

# Už sme mali – základné metódy

- Obal, ktorý umožňuje sledovať zmeny premennej
  - ▣ Vytvorenie writable signálu
    - `const count = signal(0);`
  - ▣ Čítanie - jednorazové
    - `console.log('The current count is: ' + count());`
  - ▣ Nastavovanie
    - `count.set(2);`
    - `count.update(value => value + 1);`
  - ▣ Reagovanie na zmenu (použitie v konštruktore)
    - `effect(() => console.log('Count = ' + count()));`
    - Použitie v konštruktore
  - ▣ Závislý signál (vznikne read only signál)
    - `const doubleCount = computed(() => count() * 2);`

# linkedSignal

- Kombinácia computed signálu a prepisovateľného signálu
  - ▣ `const selectedItem = linkedSignal(() => this.items()[0])`
  - ▣ Viem spustiť ručnú zmenu, ktorá zmení vypočítanú hodnotu na akú chcem
    - `selectedItem.set(myItem)`
  - ▣ Ale ak sa zmení hodnota signálu `this.items`, znova sa zmení aj `selectedItem` na prvý item

# toObservable, toSignal

- Funkcie na preklad z asynchrónneho sveta na synchronny a späť
- `prud$ = toObservable(signal);`
  - ▣ Ak sa zmení hodnota signálu, odchádza nová hodnota do prúdu
- `vysledokS = toSignal(prud$);`
  - ▣ S každou hodnotou, ktorá príde z prúdu sa zmení signál a teda sa následne spustia aj jeho prípadné listenery
  - ▣ Signál sám robí subscribe aj prestane počúvať, keď sa prúd zavrie (zachová si poslednú prijatú hodnotu)
  - ▣ Ak prúd prenesie error, ten sa vyhodí do konzoly
    - Lepšie je dávať iba také prúdy, ktoré errorry nikdy nevyhodia

# Signály ako náhrady staršej syntaxe

- Niektoré koncepty, ktoré sme sa učili už majú signálové alternatívy cca od leta 2024
- `@ViewChild(MatPaginator) pag: MatPaginator;`
  - ▣ `paginatorS = viewChild.required<MatPaginator>(MatPaginator);`
  - ▣ Nad'alej platí, že sa naplní až v čase `AfterViewInit`
- `@Input("editedUser") user = new User();`
  - ▣ `userS = input<User>(new User(), { alias: "editedUser" });`
  - ▣ `editedUser = input.required<User>();`
  - ▣ Už netreba `ngOnChanges`, stačí nové listenery na signál
- `@Output() savedUser = new EventEmitter<User>();`
  - ▣ `savedUserS = output<User>();`
  - ▣ `this.savedUserS.emit(this.myUser);`

# model signál – input a output v jednom

## Rodič

```
<app-dieta [(user)]="myUser" />
```

```
export class RodicComponent {  
  myUser = signal<User>({  
    name: "Peter",  
    email: "",  
  });  
}
```

## Dieťa

```
export class DietaComponent {  
  user = model<User>();  
  
  changeName(newName: string) {  
    this.user.update((user) => {  
      if (!user) return;  
      user.name = newName;  
      return user;  
    });  
  }  
}
```

Namiesto signálu to môže byť aj obyčajný objekt podobne ako pri template-driven formulároch



# model signál – input a output v jednom

## Alternatívny rodič

```
<app-dieta [user]="myUser" (userChange)="change($event)" />
```

```
export class RodicComponent {  
  myUser = {  
    name: "Peter",  
    email: "",  
  }  
  
  change(user: User) {  
    console.log("user sa zmenil", user);  
  }  
}
```

# rxResource a resource (typu ResourceRef)

- Obálka pre signály, ktorých hodnota sa mení na základe asynchrónneho zdroja dát (napr. zo servera)
- `const entitaResource = rxResource({  
 request: () => myQuerySignal(),  
 loader: ({request: myQuery}) =>  
 this.http.get<Entita>('/entita/' + myQuery)  
});`
- Ak sa udeje zmena v request funkcii, teda v signáli myQuerySignal, spustí sa loader s hodnotou tohto signálu a výsledok prúdu sa po čase uloží v entita

# rxResource a resource (typu ResourceRef)

- Signály, ktoré resource poskytuje
  - **value** – writable signál obsahujúci aktuálnu hodnotu
  - **isLoading** – signál (boolean) obsahujúci, či sa práve deje loader
  - **error** – signál obsahujúci chybový objekt, ak cez prúd prišla chyba
  - **status** – signál so stavom, možné hodnoty
    - ResourceStatus.Idle
    - ResourceStatus.Loading
    - ResourceStatus.Error
    - ResourceStatus.Resolved
    - ResourceStatus.Reloadng
    - ResourceStatus.Local, ak hodnota value bola nastavená lokálne

# httpResource (od Angularu 1.9.2)

- Špecializácia rxResource, ktorá na pozadí používa http klienta
- Určená iba na prijímanie dát, nie odosielanie
- `const myQuerySignal = signal<string>('desc');`
- `const entitaResource = httpResource<Entita>(() => `/entita?sort=${this.myQuerySignal()}`);`
- Ak sa udeje zmena v myQuerySignal, spustí sa nový request na server.
- Ak sa signál zmení skôr ako dôjde hodnota z prechádzajúceho requestu, starý prúd sa zruší a začne sa nový

# httpResource (od Angularu 1.9.2)

- Zvláda aj zložitejšie requesty, napr:
- `sortedUsersResource = httpResource<User[]>(() => ({  
 url: `/users`,  
 params: { sort: this.sortOrder() },  
 headers: new HttpHeaders({ 'X-My-Header': this.customHeader() })  
}));`
- Vie prijať aj iné dáta ako jsony (a meniť ich na objekty)
  - ▣ `httpResource.arrayBuffer(url, options)`
  - ▣ `httpResource.blob(url, options)`
  - ▣ `httpResource.text(url, options)`

# httpResource – dostupné hodnoty

```
interface HttpResourceRef<T> extends WritableResource<T> {
  readonly headers: Signal<HttpHeaders | undefined>;
  readonly statusCode: Signal<number | undefined>;
  readonly progress: Signal<HttpProgressEvent | undefined>;
  hasValue(): boolean;
  destroy(): void;
  readonly value: WritableSignal<T>;
  set(value: T): void;
  update(updater: (value: T) => T): void;
  asReadOnly(): Resource<T>;
  readonly status: Signal<ResourceStatus>;
  readonly error: Signal<unknown>;
  readonly isLoading: Signal<boolean>;
  reload(): boolean;
}
```

# Angular knižnice

- Vytvoríme samostatný javascriptový balíček
  - ▣ Je možné ho poslať do npm úložiska (treba vymyslieť unikátne meno)
  - ▣ A následne importovať do ľubovoľného angular projektu cez npm i balíček
- V hlavnom adresári projektu vytvoríme knižnicu
  - ▣ ng g library upjs-message
  - ▣ Chceme vypisovať pozitívne a negatívne hlášky a spracovávať http chyby
    - Použijeme komponent SnackBar

# Čo knižnica obsahuje

- `src/public-api.ts`
  - ▣ Export všetkého, čo chceme nechať používať používateľov našej knižnice
    - Komponenty, servisy, moduly,...
  - ▣ To, čo sa neexportuje, zvonku nie je vidieť
- `src/lib/*`
  - ▣ Naše zdrojové súbory, ktoré naprogramujeme
- `package.json`
  - ▣ V `peerDependencies` uvedieme závislosť na knižniciach
  - ▣ Vieme upraviť názov a verziu knižnice
- `README.md`
  - ▣ Informácie, ktoré sa zverejnia ako popis na portáli `npmjs.com`



# Ako knižnicu použiť

- Ak chceme použiť knižnicu v projekte musíme ju vybuildovať
  - ▣ `ng build kasv2024-message`
- Potom vieme normálne importovať do modulu alebo komponentu v sekcii `imports` všetko, čo sme exportovali
- Export do npm úložiska z `dist` adresára knižnice
  - ▣ `npm adduser váš_login`
  - ▣ `npm publish`