



ANGULAR – ADVANCED

Peter Gurský, peter.gursky@upjs.sk

Zdrojáky z úvodného kurzu

- `https://github.com/PeterGursky/elct-angular-2022-05`
- Klonovanie projektu
 - ▣ `git clone https://github.com/PeterGursky/elct-angular-2022-05.git`
 - ▣ `cd elct-angular-2022-05`
 - ▣ `npm i`
 - `java -jar films.server.jar`
 - `ng serve --port=4201`
- My si však vytvoríme aj nový projekt
 - ▣ `ng new elct-angular-2022-08`
 - ▣ `cd elct-angular-2022-08`
 - `ng serve`

Angular Material

- <https://material.angular.io/>
- `ng add @angular/material`
- pre každý komponent knižnice je potrebné importovať do projektu príslušný modul do nášho aplikačného modulu (aktuálne `app.module.ts`)
 - ▣ treba pozrieť dokumentáciu komponentu – sekcia API

Login konponent

- Spravíme si aplikáciu, ktorá bude prístupná iba prihláseným používateľom
 - ▣ Po prihlásení zobrazíme zoznam používateľov
- Využijeme z minulého projektu `UserService` a `MessageService`
 - ▣ Prerobíme `MessageService` pomocou komponentu `SnackBar`
 - môžeme si prispôbiť farby cez vlastnosť `panelClass` a nastavenie `css`

Navigačná lišta

- Využijeme material komponent toolbar
 - ▣ potrebujeme aj komponenty button a icon
- V navigácii dáme linky na všetky zatiaľ používané URL adresy

Feature Module

- Keď aplikácia rastie, rozdelíme ju na rôzne zamerané časti, ktoré spolupracujú s koreňovým modulom

ng generate module users --routing

- vznikne súbor users.module.ts

```
import { NgModule } from '@angular/core';  
import { CommonModule } from '@angular/common';  
import { UsersRoutingModule } from './users-routing.module';
```

```
@NgModule({  
  declarations: [],  
  imports: [CommonModule, UsersRoutingModule]  
})  
export class UsersModule {}
```

podmnožina BrowserModule,
ktorý sa používa v koreňovom
module

Komponent v module

- komponent pridávame do adresára s rovnakým názvom ako modul

ng generate component users/users

- v module users.module.ts sa doplní:

```
import { UsersComponent } from './users/users.component';

@NgModule({
  declarations: [UsersComponent],
  imports: [CommonModule, UsersRoutingModule]
})
export class UsersModule {}
```

Importovanie modulu

app.module.ts

```
import { NgModule } from '@angular/core';  
import { UsersModule } from '../users/users.module';
```

```
@NgModule({  
  ...  
  imports: [  
    ...  
    UsersModule,  
    AppRoutingModule,  
    ...  
  ],  
  ...  
})  
export class AppModule { }
```

Musíme importovať pred hlavným routovaním, aby najprv vyhodnocovalo trasy modulu a až potom testovalo všetko ostatné, napr. '**'

Export komponentov

- ak by sme chceli komponenty z modulu použiť priamo v komponentoch inej časti aplikácie, musíme ich z modulu exportovať, doplníme:

users.module.ts

```
@NgModule({  
  declarations: [UsersComponent],  
  imports: [CommonModule, UsersRoutingModule],  
  exports: [UsersComponent ]  
})  
export class UsersModule { }
```

Export komponentov

- ak by sme chceli komponenty z modulu použiť priamo v komponentoch inej časti aplikácie, musíme ich z modulu exportovať, doplníme:

users.module.ts

```
@NgModule({  
  declarations: [UsersComponent],  
  imports: [CommonModule, UsersRoutingModule],  
  exports: [UsersComponent]  
})  
export class UsersModule { }
```

Nás to teraz trápiť nemusí, my necháme naťahovanie komponentov z modulu na modulový router.

Routovanie modulov

- vytvoríme vlastnú konfiguráciu routovania pre modul
- už máme vytvorený súbor `users-routing.module.ts`
 - ▣ v ňom robíme to isté ako v hlavnom len namiesto `RouteModule.forRoot(..)` píšeme `RouteModule.forChild(..)`

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

const routes: Routes = [];

@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class UsersRoutingModule { }
```

Material table

- Minimalistický príklad:
 - ▣ `mat-text-column` – len pre reťazcové premenné
 - ▣ `matHeaderRowDef` – na iterovanie stĺpcov podľa parametra **name** v `mat-text-column` – číta **headerText**
 - ▣ `matRowDef` – na iterovanie riadkov, číta `row['name']` a `row['email']`

inštančná premenná `users=[
{name: 'Jano', email: 'j@i.sk'},
{name: 'Fero', email: 'f@i.sk'}];`

```
<table mat-table [dataSource]="users">  
  <mat-text-column name="name" headerText="Name"></mat-text-column>  
  <mat-text-column name="email" headerText="E-mail"></mat-text-column>  
  
  <tr mat-header-row *matHeaderRowDef="columnsToDisplay"></tr>  
  <tr mat-row *matRowDef="let row; columns: columnsToDisplay"></tr>  
</table>
```

`columnsToDisplay=['name', 'email']`

Material table

- jednoduché získanie dát zo servera
- implementujeme interface `AfterViewInit` – aby sme si zvykli, keď budeme používať `MatPaginator` resp. `MatSort` – kde je to žiadúce

```
users = [];  
  
constructor(private userService: UserService) {}  
  
ngAfterViewInit() {  
    this.userService  
        .getExtendedUsers()  
        .subscribe(users => (this.users = users));  
}
```

Zložitejšie stĺpce

ng-container sa nestane
elementom v DOM

```
<ng-container matColumnDef="lastLogin">
  <th mat-header-cell *matHeaderCellDef>Last login</th>
  <td mat-cell *matCellDef="let user">
    {{ user.lastLogin | date: 'd.M.y H:mm:ss' }}
  </td>
</ng-container>
```

MatPaginator

- Komponent na zobrazenie paginácie
- Keď používateľ interaguje s komponentom, generuje cez výstupný prúd 'page' objekt typu PageEvent
- Na nastavenie komponentu je možné použiť viacero vstupných parametrov

```
<mat-paginator  
  [length]="users.length"  
  [pageIndex]="0"  
  [pageSize]="10"  
  [pageSizeOptions]="[2, 5, 10, 20]"  
></mat-paginator>
```

MatTableDataSource<Entita>

- Zdroj statických dát pre tabuľku
- Má vstavanú podporu pre získavanie vstupov z komponentov MatPaginator a MatSort
- Má podporu pre filtrovanie riadkov pomocou reťazcového filtra
 - ▣ Filter sa nastaví cez premennú filter
 - ▣ Výsledné dáta zodpovedajúce filtru vieme aj získať cez premennú filteredData

Použitie paginátora

- Potrebujeme získať referenciu na detský komponent paginátora do premennej v komponente
 - ▣ `@ViewChild(MatPaginator) paginator: MatPaginator;`
- Poskytneme túto referenciu pre `MatTableDataSource` v `ngAfterViewInit()`
 - ▣ `this.dataSource.paginator = this.paginator;`

Filtrovanie

- Na filtrovanie nemáme žiaden špeciálny komponent, použijeme obyčajný input a reagujeme na udalosť keyup
- daný reťazec potom vložíme do MatTableDataSource cez premennú filter
 - ▣ `this.dataSource.filter = filterValue`
 - ▣ je fajn paginátoru povedať nech sa potom hneď presunie na prvú stránku
 - `this.dataSource.paginator.firstPage();`
- Ak chceme vlastnú implementáciu filtrovania nasetujeme do MatTableDataSource funkciu v tvare `((data: T, filter: string) => boolean)` cez premennú filterPredicate

Parametre URL:

- ak chceme poslať povinný parameter napr. pre URL adresu `http://localhost:4200/users/25`
 - `<a [routerLink]="['/users', 25]">`
 - `app-routing.module.ts` doplníme
 - `{ path: '/users/:id', component: UserDetailsComponent }`
 - v `UserDetailsComponent` :

```
import { ActivatedRoute } from '@angular/router';  
  
export class UserDetailsComponent implements OnInit {  
  
  constructor(private route: ActivatedRoute) {}  
  
  ngOnInit() {  
    this.userId = this.route.snapshot.params['id'];  
  }  
}
```

táto syntax je (zatiaľ)
ok, ale iba ak sa tento
komponent už nebude
používať inde

Parametre URL:

- ak sa komponent znovupoužíva (nerenderujeme ho celý), napr. ak z jedného používateľa (rovnaká trasa) sa vieme prekliknúť na iného, počúvame na zmeny parametra:

```
import { ActivatedRoute, ParamMap } from '@angular/router';

ngOnInit() {
  this.route.paramMap.pipe(
    map((params: ParamMap) => params.get('id'))
  ).subscribe(userId => this.userId = userId);
}
```

Parametre URL:

- ak sa komponent znovupoužíva (nerenderujeme ho celý), napr. ak z jedného používateľa (rovnaká trasa) sa vieme prekliknúť na iného, počúvame na zmeny parametra:

```
import { ActivatedRoute, ParamMap } from '@angular/router';
import { switchMap } from 'rxjs/operators';

ngOnInit() {
  this.route.paramMap.pipe(
    switchMap((params: ParamMap) =>
      this.service.getUser(+params.get('id')))
  ).subscribe(user => this.user = user);
}
```

Parametre URL:

- ak sa komponent znovupoužíva (nerenderujeme ho celý), napr. ak z jedného používateľa (rovnaká trasa) sa vieme prekliknúť na iného, počúvame na zmeny parametra:

```
import { ActivatedRoute, ParamMap } from '@angular/router';
import { switchMap } from 'rxjs/operators';

ngOnInit() {
  this.route.paramMap.pipe(
    switchMap((params: ParamMap) =>
      this.service.getUser(+params.get('id')))
  ).subscribe(user => this.user = user);
}
```

Funguje ako mergeMap s tým, že ak sa ešte čaká na ukončenie predošlého service.getUser(..), je to zrušené a začne sa vykonávať nové

Parametre URL:

- ak sa komponent znovupoužíva (nerenderujeme ho celý), napr. ak z jedného používateľa (rovnaká trasa) sa vieme prekliknúť na iného, počúvame na zmeny parametra:

```
import { ActivatedRoute, ParamMap } from '@angular/router';
import { switchMap } from 'rxjs/operators';

ngOnInit() {
  this.route.paramMap.pipe(
    switchMap((params: ParamMap) =>
      this.service.getUser(+params.get('id')))
  ).subscribe(user => this.user = user);
}
```

okrem **get(parameter)**, môžeme použiť aj metódy:

- has(parameter)** – true, ak taký parameter máme
- getAll(parameter)** – ak máme pre parameter viac hodnôt
- keys()** – vráti názvy parametrov

Nepovinné parametre

- Niekedy chceme v URL poslať nepovinné parametre napr. pre pagináciu

- ▣ `http://localhost:4200/users;page=2;count=10`

matrix notácia cez ;
– nie cez ? a &

- ▣ v kóde:

- `this.router.navigate(['/users', { page: 2, count: 10 }]);`

```
import { ActivatedRoute, ParamMap } from '@angular/router';
import { switchMap } from 'rxjs/operators';
```

```
ngOnInit() {
  this.route.paramMap.pipe(
    switchMap((params: ParamMap) =>
      this.page = params.has('page') ? +params.get('page') : 1);
      this.count = params.has('count') ? +params.get('count') : 5);
      return this.service.getExtendedUsers();
    ).subscribe(users => this.users = users);
}
```


Niektoré ďalšie parametre

ActivatedRoute

- url
 - ▣ observable s poľom reťazcov častí cesty z URL v tejto trase
- data
 - ▣ observable s dátami pre danú trasu, vrátane dát získaných neskôr resolverom
 - ▣ do každej trasy vieme dať parameter data, ktorý môže obsahovať čokoľvek
- queryParamsMap
 - ▣ observable s parametrami pre všetky trasy
 - ▣ parametre v URL uvedené za otáznikom v tvare ?atr1=val1&atr2=val2
- fragment
 - ▣ observable s hodnotou fragmentu/časti stránky – v URL je to hodnota za #
 - ▣ referencuje element s daným atribútom id

Hierarchické routovanie

- pod každou trasou môžeme vytvoriť jej deti
- ak je trasa úspešná, vykreslí svoj komponent a ak na jeho koniec dodáme `<router-outlet></router-outlet>`, tak na tomto mieste umiestni komponent úspešnej trasy niektorého zo svojich detí

```
const groupsRoutes: Routes = [  
  {  
    path: 'groups',  
    component: GroupsComponent,  
    children: [  
      {  
        path: '',  
        component: GroupsListComponent,  
        children: [  
          {  
            path: ':id',  
            component: GroupDetailComponent  
          },  
          {  
            path: '',  
            component: GroupHomeComponent  
          }  
        ]  
      }  
    ]  
  }  
];
```

<http://localhost:4200/groups/2>

GroupComponent

GroupsListComponent

GroupDetailComponent

<http://localhost:4200/groups>

GroupComponent

GroupsListComponent

GroupHomeComponent

Hierarchické routovanie

- router defaultne znovupoužíva komponenty detských trás
- každé dieťa predstavuje rozšírenie URL adresy o svoju trasu
 - ▣ pozerá sa napravo od toho, čo použil z URL rodič
- deti môžu používať aj absolútne cesty – vtedy path začína s /
- pri navigovaní je možné používať aj "../" na posun o úroveň vyššie v ceste
 - ▣ napr. ak chceme z aktuálnej skupiny ísť na skupinu 5:
 - ▣ `this.router.navigate(['../', { id: 5 }], { relativeTo: this.route });`

viac možností, pre druhý parameter:

<https://angular.io/api/router/NavigationExtras>

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

- strážca implementuje aspoň jeden z interface-ov
 - ▣ CanActivateTo – stráží navigáciu na konkrétnu trasu
 - ▣ CanActivateChild – na deti v hierarchii routovania
 - ▣ CanDeactivate – stráží odchod z aktuálnej trasy
 - ▣ Resolve – spracováva dáta trasy pred aktiváciou trasy
 - ▣ CanLoad – stráží navigáciu na modul natiahnutý asynchrónne

Vytvorenie strážcu

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

```
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot, UrlTree } from
 '@angular/router';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {

  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot)
    : Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {
    return true;
  }
}
```

Vytvorenie strážcu

objekt aktivovanej (**budúcej**) trasy z routra, ktorú strážime. Máme prístup napr. ku

- spárovanému komponentu,
- vnoreným trasám aj rodičovskej trase,
- častiam URL, ktorá bola použitá
- dátam/parametrom, ktoré cez URL prišli

/auth

```
import { CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
```

```
@Injectable({  
  providedIn: 'root'  
})
```

```
export class AuthGuard implements CanActivate {
```

```
  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {  
    return true;  
  }  
}
```

```
}
```

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

Použitie strážcu

- strážca stráži celý podstrom trás
 - ▣ do podtrás sa vnoríme, iba keď strážca pustí rodiča
 - ▣ ak chceme chrániť IBA podstrom použijeme interface `CanActivateChild`
- poradie overovania:
 - ▣ najskôr sa overujú strážcovia `CanDeactivate` a `CanActivateChild` z najhlbšieho vnorenia trasy až po najvyššie,
 - ▣ potom sa overujú `CanActivate` strážcovia z najvyššej trasy po najnižšiu
 - ▣ ak ktorýkoľvek strážca vráti `false/UrlTree`, ďalší strážcovia sa neoverujú

Použitie strážcu

- sprážca stráži celý podstrom trás
 - ▣ do podtrás sa vnoríme, iba keď strážca pustí rodiča
 - ▣ ak chceme chrániť IBA podstrom použijeme interface `CanActivateChild`

```
const usersRoutes: Routes = [  
  {  
    path: 'users',  
    component: UsersComponent,  
    canActivate: [AuthGuard],  
    children: [  
      ...  
    ]  
  }  
];
```

Presmerovanie na LoginComponent

```
export class AuthGuard implements CanActivate {  
  constructor(private router: Router, private userService: UsersService) {}  
  
  canActivate(next: ActivatedRouteSnapshot, state: RouterStateSnapshot) {  
    if (this.userService.user) {  
      return true;  
    }  
    this.userService.redirectAfterLogin = state.url;  
    this.router.navigateByUrl('/login');  
    return false;  
  }  
}
```

alternatívne:

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

Strážca CanDeactivate

- 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 zistí, či komponent má metódu canDeactivate a ak áno zariadi sa podľa nej

Strážca CanDeactivate

ng g guard guards/can-deactivate

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

```
export interface CanComponentDeactivate {  
  canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;  
}  
  
@Injectable({  
  providedIn: 'root'  
})  
export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {  
  
  canDeactivate(component: CanComponentDeactivate, _currentRoute: ActivatedRouteSnapshot,  
    _currentState: RouterStateSnapshot, _nextState: RouterStateSnapshot)  
    : Observable<boolean> | Promise<boolean> | boolean {  
    return component.canDeactivate ? component.canDeactivate() : true;  
  }  
}
```

Príklad chráneného komponentu

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

  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: UsersRestClientService) {}

  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 || 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);
  }
}
```

V trase zaevidujeme strážcu:

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

Resolve – získanie dát pred aktiváciou trasy

- ak chceme pred renderovaním stránky získať/pripraviť dáta, aby sa už zobrazil iba finálny komponent
 - ▣ ak nastane chyba, môžem sa rovno preroutovať inde a nevytvárať komponent
- Postup:
 - ▣ vytvoríme službu implementujúcu **Resolve** 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, alebo nevráti dáta, router presmerujeme inam, napr. naspäť

Resolver na získanie konkrétnej skupiny

```
export class GroupDetailResolverService implements Resolve<Group> {  
  constructor(private router: Router, private userService: UsersService) {}  
  
  resolve(route: ActivatedRouteSnapshot, _state: RouterStateSnapshot)  
    : Observable<Group> | Observable<never> {  
    return this.userService.getGroup(+route.paramMap.get('id'));  
  }  
}
```


Komponent a trasa

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

  constructor(private route: ActivatedRoute) {}

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

V trase namapujeme resolver na premennú group:

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

Asynchrónne routovanie

- Umožňuje natiahnuť iba úvodné komponenty a potom zvyšné moduly
 - ▣ buď natiahnuť na pozadí pokiaľ používateľ už pracuje s úvodnými komponentami
 - ▣ alebo ich dotiahnuť až po navigácii na príslušnú URL
- Aj veľké aplikácie pracujú svižne
 - ▣ nemá zmysel naťahovať niečo, čo sa nepoužije, napr. ak používateľ ani nemá práva pre danú sekciu
 - ▣ ideálne natiahnem len to, čo aj použijem
 - ▣ tie časti, kde pravdepodobne používateľ pôjde neskôr, netreba spracovať pred prvým klikom do stránky, no budú už pravdepodobne pripravené, keď tam neskôr pôjde

Lazy loading

- naťahovanie modulu na požiadanie
- neimportujem ho v app.module.ts
- importujem do v trase v app-routing.module.ts:

```
{  
  path: 'group',  
  loadChildren: () =>  
    import('./groups/groups.module').then(mod => mod.GroupsModule)  
}
```

- v groups-routing.module.ts nahradím koreňovú cestu prázdny reťazcom

Strážca CanLoad

- použije sa v trase, kde sa robí loadChildren

```
{  
  path: 'group',  
  loadChildren: () => import('./groups/groups.module').then(mod =>  
    mod.GroupsModule),  
  canLoad: [AuthGuard]  
}
```

- metódu canLoad píšeme analogicky ako canActivate
 - ▣ vracia boolean | Observable<boolean> | Promise<boolean>
 - true ak chceme naložovať,
 - false ak nechceme – tiež môžeme pred vrátením false prenavigovať inam
 - ▣ na vstupe metódy canLoad 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 CanLoad
 - PreloadAllModules – stratégia vyberá každý modul
 - vlastná stratégia – implementujúca PreloadingStrategy

nastavenie preloadingStrategy

- pridáme parameter do metódy forRoot() hlavného routra

```
import { RouterModule, Routes, PreloadAllModules } from '@angular/router';

const routes: Routes = [
  ...
];

@NgModule({
  imports: [RouterModule.forRoot(routes, {
    preloadingStrategy: PreloadAllModules
  })],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

Vlastná stratégia

- dodáme si do trasy vlastné dáta

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

- vyrobíme vlastnú stratégiu, v ktorej automaticky natiahneme na pozadí iba také moduly, ktoré majú v dátach trasy `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 trasami
 - ▣ <https://angular.io/guide/router#adding-routable-animations>
- paralelné routovanie v pomenovaných outletoch
 - ▣ <https://angular.io/guide/router#displaying-multiple-routes-in-named-outlets>

Validátory

- Angular má niekoľko vlastných validátorov na kontrolu formulárov, alebo môžeme dorobiť vlastné
- Neparametrické validátory sú funkcie (ValidatorFn), ktoré majú
 - ▣ na vstupe kontrolovaný formulárový element, skupinu formulárových elementov alebo celý formulár
 - ▣ na výstupe
 - null ak je validátor úspešný
 - objekt s nájdenými chybami, napr. {'length': 'small string', 'format': 'wrong character', }
- Parametrické validátory sú funkcie druhého rádu
 - ▣ na vstupe majú jeden alebo viac hodnôt na ich nakonfigurovanie (napr. Validator.minLength(5))
 - ▣ na výstupe má ValidatorFn

Vstavané validátory

- <https://angular.io/api/forms/Validators>
 - ▣ parametrické
 - min, max, minLength, maxLength, pattern, compose, composeAsync
 - ▣ neparametrické
 - required, requiredTrue, email, nullValidator

Template-driven formuláre

- založené na
 - ▣ [(ngModel)] na mapovanie hodnôt
 - ▣ povinný parameter name
- model formulára, ktorý vyhodnocuje validitu, je dostupný **len v šablóne** cez premennú elementu
 - ▣ ngForm v elemente form
 - ▣ ngModel vo vstupných formulárových komponentoch
- model formulára nie je dostupný z kódu komponentu
- ak chceme použiť vo formulári validátory, vieme ich dodať formulárovým elementom iba cez direktívy, t.j. atribúty elementov
 - ▣ <https://angular.io/guide/form-validation#adding-to-template-driven-forms>

Vloženie validátora pre TD-formuláre

- direktívy pre zabudované validátory:
 - <https://angular.io/api/forms#directives>
 - CheckboxRequiredValidator, PatternValidator,...
- Napr. EmailValidator je direktíva definovaná pravidlami:
 - [email][formControlName] ← pre Reakt. formuláre
 - [email][formControl] ← pre Reakt. formuláre
 - [email][ngModel] ← pre TD-formuláre
- takže ho aktivujeme tak, že napíšeme v šablóne
 - <input type="email" name="email" **ngModel email**>
 - <input type="email" name="email"
[(**ngModel**)]="user.email" **email**>

Použitie validácie

- validita sa dá zobrazit' používateľovi pomocou css, ak je prítomný validátor nastaví sa pre daný formulárový element jedna z tried
 - ▣ ng-touched / ng-untouched – element bol navštívený / nebol
 - ▣ ng-dirty / **ng-pristine** – hodnota je zmenená / nie je
 - ▣ **ng-valid** / **ng-invalid** - hodnota je správna / nie je
 - ak element má atribút required, hodnota musí byť vyplnená
 - okrem required máme aj: minlength, maxlength, pattern
 - ...zložitejšie kontroly vid'. Validator v ngModel-i
- Pozrime si cez inšpektora

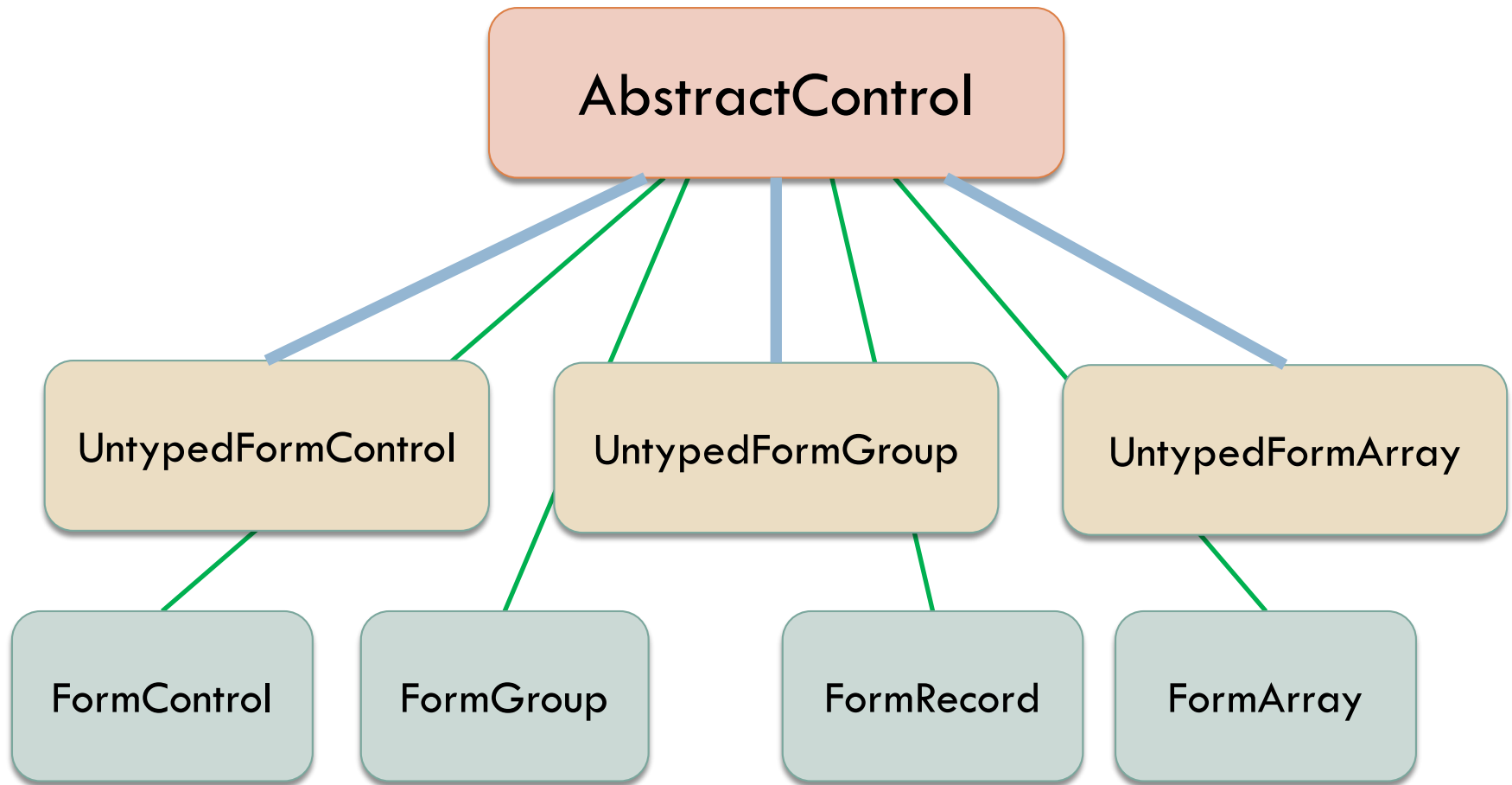
Validita cez Angular Material

- nevalidný input je označený červenou farbou
- mat-form-field môže mať v sebe okrem input elementu aj
 - ▣ <mat-hint> - zobrazenie textu pod input elementom
 - ▣ <mat-label> - pomenovanie vstupu
 - ak je input prázdny zobrazuje sa namiesto hodnoty
 - ak je input vyplnený zobrazuje sa nad hodnotou
 - ▣ <mat-error> - zobrazenie červenej chybovej hlášky pod input elementom
 - použijeme *ngIf, aby sa chyba zobrazila, iba ak je input nevalidný
 - ak sa zobrazuje mat-error, nezobrazuje sa mat-hint

Reaktívne formuláre

- ak ich chceme používať
 - ▣ importujeme `ReactiveFormsModule` do `@NgModule`
- model formulára a jeho komponentov sú vytvárané ako (inštančné) premenné v kóde komponentu
 - ▣ formulár alebo skupina komponentov je typu `FormGroup`
 - ▣ jeden komponent (napr. `<input>`) je typu `FormControl`
- hodnota komponentu formulára nie je prepojená s inštančnou premennou, ako v TD-formulároch
 - ▣ nepoužívame `[(ngModel)]="username"` pre inštančnú premennú `username='Jano'`,
 - ▣ hodnota komponentov sa dá z kódu meniť len cez funkcie `setValue()`, `patchValue()` – vid' neskôr

Hierarchia reaktívnych komponentov



Prepojenie šablóny a reaktívneho modelu

- samostatné elementy – bez nadradeného `<form>` elementu:
 - v šablóne: `<input type="text" [formControl]="name">`
 - v kóde: `name = new FormControl<string | null>('Jano');`
- pre skupinu elementov:

```
<form [formGroup]="profileForm">  
  <input type="text" formControlName="firstName">  
  <input type="text" formControlName="lastName">  
</form>
```

```
profileForm = new FormGroup({  
  firstName: new FormControl(""),  
  lastName: new FormControl(""),  
});
```

```
constructor(private fb: FormBuilder) {  
  profileForm = this.fb.group({  
    firstName: [], lastName: []  
  });  
}
```

Prepojenie šablóny a reaktívneho modelu

úvodné hodnoty

- samostatné elementu:

- v šablóne: `<input type="text" [formControl]="name">`
- v kóde: `name = new FormControl<string | null>('Jano');`

- pre skupinu elementov:

```
<form [formGroup]="profileForm">  
  <input type="text" formControlName="firstName">  
  <input type="text" formControlName="lastName">  
</form>
```

```
profileForm = new FormGroup({  
  firstName: new FormControl(""),  
  lastName: new FormControl(""),  
});
```

```
constructor(private fb: FormBuilder) {  
  profileForm = this.fb.group({  
    firstName: [], lastName: []  
  });  
}
```

Prepojenie šablóny a reaktívneho modelu

- samostatné elementy – bez nadradeného elementu:
 - v šablóne: `<input type="text" [formControl]="name">`
 - v kóde: `name = new FormControl<string | null>('Jano');`

Typ možných hodnôt uvádzame v zobáčkoch

Typ písať nemusíme, ak je odvoditeľný z úvodnej hodnoty (okrem booleanu – ten píšeme vždy)

Odvodí sa typ `<string | null>`

to:

```
">  
Name="firstName">  
Name="lastName">
```

```
profileForm = new FormGroup({  
  firstName: new FormControl(""),  
  lastName: new FormControl(""),  
});
```

```
constructor(private fb: FormBuilder) {  
  profileForm = this.fb.group({  
    firstName: [], lastName: []  
  });  
}
```

Prepojenie šablóny a reaktívneho modelu

Typ možných
hodnôt uvádzame
v zátvorkách

Ak nechceme null hodnoty vo `FormControl<string | null>`, musíme uviesť náš zámer cez druhý parameter konštruktora:

```
firstName: new FormControl<string>("", {nonNullable: true})
```

alebo použiť `fb: FormBuilder`: `firstName = this.fb.nonNullable.control("");`

alebo použiť `nnfb: NonNullableFormBuilder`: `firstName = this.nnfb.control("");`

Ak potom zavoláme `firstName.reset()`, nenastaví hodnotu na null, ale na iniciálnu hodnotu.

```
profileForm = new FormGroup({  
  firstName: new FormControl(""),  
  lastName: new FormControl(""),  
});
```

```
constructor(private fb: FormBuilder) {  
  profileForm = this.fb.group({  
    firstName: "", lastName: ""  
  });  
}
```

Prepojenie šablóny a reaktívneho modelu

□ pre pole elementov:

```
<form [formGroup]="profileForm">
  ...
  <div formArrayName="aliases">
    <input *ngFor="let alias of aliases.controls; let i=index"
      type="text" [formControlName]="i">
  </div>
</form>
```

```
profileForm = new FormGroup({
  ...
  aliases: new FormArray([
    new FormControl('Johnny'),
    new FormControl('Janči')
  ]),
});
```

```
constructor(private fb: FormBuilder) { }
profileForm = this.fb.group({ ...,
  this.fb.array([
    this.fb.control(""),
    this.fb.control("")
  ])
});
```

Prepojenie šablóny a reaktívneho modelu

□ pre pole elementov:

```
<form [formGroup]="profileForm">
  ...
  <div formArrayName="aliases">
    <input *ngFor="let alias of aliases"
      type="text" [formControl]="alias.control" />
  </div>
</form>
```

Pridanie nového aliasu:

```
<button (click)="addAlias()">Add Alias</button>
```

```
profileForm = new FormBuilder()
...
aliases: new FormArray([
  new FormControl(''),
  new FormControl('')
]),
});
```

```
get aliases() {
  return this.profileForm.get('aliases') as FormArray;
}

addAlias() {
  this.aliases.push(this.fb.control(''));
}
```

Hodnoty komponentov r. formulárov

- získanie hodnoty samostatného elementu
 - ▣ `surname = new FormControl('Pekný');`
 - ▣ v šablóne: `{{surname.value}}`
 - ▣ `let currentName = this.surname.value;`
 - ▣ `this.surname.valueChanges.subscribe(value => x = value);`
- získanie hodnoty vnoreného elementu

```
profileForm = new FormGroup({  
  person = new FormGroup({  
    name: new FormControl(""),  
    surname: new FormControl("")  
  })  
});
```

- ▣ `let currentFirstName = this.profileForm.get('person.name').value`
- ▣ v šablóne: `{{name.value}}`
 - ak máme getter: `get name() { return this.profileForm.get('name'); }`

Zmena hodnoty r. formulárov

- hodnoty sú immutable
 - ▣ nastavíme všetky hodnoty formulára cez setValue()
 - ▣ nastavíme iba niektoré hodnoty cez patchValue()
 - platí pre ľubovoľnú úroveň

```
profileForm = new FormGroup({  
  firstName: new FormControl(""),  
  lastName: new FormControl(""),  
  address: new FormGroup({  
    street: new FormControl(""),  
    city: new FormControl(""),  
    state: new FormControl(""),  
    zip: new FormControl("")  
  })  
});
```

```
updateProfile() {  
  this.profileForm.patchValue({  
    firstName: 'Nancy',  
    address: {  
      street: '123 Drew Street'  
    }  
  });  
}
```

Validátory pre r. formuláre

- validátory dodávame ako druhý parameter konštruktora, asynchrónne validátory ako tretí
 - ▣ `name = new FormControl(user.name, Validators.required);`
 - ▣ `email = new FormControl(user.email, [Validators.required, Validators.email], MyAsyncValidator);`
- alternatívne ako objekt cez druhý parameter konštruktora, napr.:
 - ▣ `myForm = new FormGroup({ 'name': new FormControl(), 'email': new FormControl() }, { validators: myFormValidator, asyncValidators: [FirstValidator, SecondValidator] });`

Výsledok validity vo formulári

required nie je
povinné zadať, ale
je to odporúčané

```
<input id="name" class="form-control" [formControl]="name" required >
```

```
<div *ngIf="name.invalid && (name.dirty || name.touched)" class="danger">
```

```
<div *ngIf="name.errors.required">
```

Name is required.

```
</div>
```

```
<div *ngIf="name.errors.minlength">
```

Name must be at least 4 characters long.

```
</div>
```

```
</div>
```

```
name = new  
FormControl(this.user.name, [  
  Validators.required,  
  Validators.minLength(4)]);
```

d'alšie vlastnosti vid': <https://angular.io/api/forms/AbstractControl>

Vlastný validátor pre formulárový komponent

- vytvoríme si validátor na kvalitu hesla cez knižnicu zxcvbn
 - ▣ npm i zxcvbn
 - ▣ npm i @types/zxcvbn --save-dev

```
passwordValidator(): ValidatorFn {  
  return (control: AbstractControl): ValidationErrors => {  
    const test = zxcvbn(control.value);  
    const message = 'Password strength: '  
      + test.score + ' / 4 - must be 3 or 4';  
    return test.score < 3 ? { weakPassword: message } : null;  
  };  
}
```

Vlastný validátor pre formulár

- vytvoríme si validátor na zhodu hesla a kontrolného hesla

```
passwordsMatchValidator(control: FormGroup): ValidationErrors {  
  const password = control.get('password');  
  const password2 = control.get('password2');  
  if (password.value === password2.value) {  
    password2.setErrors(null);  
    return null;  
  } else {  
    password2.setErrors({ differentPasswords: 'Passwords do not  
match' });  
    return { differentPasswords: 'Passwords do not match' };  
  }  
}
```

Vlastný asynchrónny validátor

- vytvoríme si validátor na konflikt s menom alebo emailom pri registrácii
- použijeme na serveri endpoint `/user-conflicts`
 - ▣ vracia pole s názvami konfliktných premenných, možné hodnoty sú "email" a "name", alebo vráti prázdne pole ak konflikt nie je

Filmy

- náš REST server umožňuje CRUD operácie nad lokálnou DB filmov
- <http://localhost:8080/films> má voliteľné parametre na server-side pagináciu, usporiadanie a filtrovanie
 - ▣ `Optional<String> orderBy`
 - možné: `nazov`, `slovenskyNazov`, `rok`, `poradieVRebricku.AFI 1998`, `poradieVRebricku.AFI 2007`
 - ▣ `Optional<Boolean> descending`,
 - ▣ `Optional<Integer> indexFrom`,
 - ▣ `Optional<Integer> indexTo`,
 - ▣ `Optional<String> search`

DataSource<Entita>

- Interface, ktorý vynucuje 2 metódy
 - ▣ connect(): Observable<Entita[]>
 - Tabuľka sa updatuje, keď výstupný prúd emituje novú hodnotu
 - ▣ disconnect()
- V konštruktore implementujúcej triedy si vypýtame, čo potrebujeme v metóde connect
- Inštancia datasource-u je modelom pre tabuľku
 - ▣ <table mat-table [dataSource]="datasource">

Websockets

- Udržiavaný obojsmerný kanál na komunikáciu so serverom
 - ▣ nie len request-response ako pri bežnom HTTP
- cieľom je:
 - ▣ poskytnúť funkcionality TCP spojení cez HTTP protokol s využitím jeho šifrovania a bezpečnosti cez origin a prípadného proxy cez ten istý port ako bežná HTTP komunikácia
 - ▣ umožniť fungovať viacerým službám cez pomenovania
 - ▣ komunikácia cez rámce bez obmedzenia veľkosti
 - ▣ korektné ukončenie spojenia

Inštalácia

- Websocket je súčasťou javascriptu, ale príliš nízkoúrovňový (ako TCP)
 - ▣ potrebujeme nadstavbový protokol, ktorý označuje, ktorá správa je aká
 - ▣ STOMP = Streaming Text Oriented Messaging Protocol
 - správy CONNECT, DISCONNECT, SUBSCRIBE, UNSUBSCRIBE, BEGIN, SEND, COMMIT, ABORT, ACK, NACK
- nainštalujeme knižnice
 - ▣ npm install stompjs net
 - ▣ npm install @types/stompjs --save-dev

Použitie

□ napojenie:

- **socket**: `WebSocket = new WebSocket('ws://localhost:8080/ws');`
- `stompClient`: `Stomp.Client = Stomp.over(socket);`
- `this.stompClient.connect(
 header, (frame:Stomp.Frame)=>{}, error=>{});`
 - druhý parameter je callback funkcia, zavolaná, keď príde nový rámec s informáciou o úspešnom pripojení
- po napojení je spojenie vytvorené a
 - môžeme sa urobiť subscribe na dané topic-y message brokera
 - posielat' správy vybranému cieľu

□ odpojenie:

- `socket.close()`

počúvanie topic-u a posielanie správ

- `this.stompClient.subscribe('/topic/messages', msg => { .. });`
 - ▣ prvý parameter je meno topic-u na serveri z ktorého prichádzajú správy
 - ▣ druhý parameter je callback keď príde správa
 - ▣ tretí, nepovinný je hlavička

- `this.stompClient.send('/app/hello', {}, 'hello world');`
 - ▣ prvý parameter je cieľ, kam sa posiela správa
 - ▣ druhý parameter je hlavička
 - ▣ tretí parameter je samotná správa

NGXS

- manažovanie stavu aplikácie na jednom mieste
 - ▣ stav = pole modelov aplikačnej domény
 - ▣ stav je dostupný všade
- motivácia z Redux-u známeho z React aplikácií
 - ▣ ide však ďalej – lepšia integrácia s Angularom
 - anotácie, dependency injection, ...
- základná idea
 - ▣ aplikácia/používateľ vyvoláva (dispatch) udalosti – akcie
 - ▣ akcia spôsobuje reakcie (spustenie reducera)
 - vykonanie kódu – napr. komunikáciu so službami
 - zmena stavu / modelu v stave
 - ▣ zmenený stav môžeme čítať (select)
 - synchrónne – pýtam sa na aktuálnu hodnotu - snapshot
 - asynchrónne – cez Observable/Promise

NGXS - inštalácia

- nainštalujeme balíček
 - ▣ npm i @ngxs/store --save
- vytvoríme stavové triedy (vid'. ďalej), napr. majme triedu *AuthState*
 - ▣ doplníme do importov v app.module.ts:
 - selectorOptions nebudú potrebné v NGXS ver.4

```
@NgModule({
  declarations: [ ... ],
  imports: [
    NgxsModule.forRoot([AuthState], {
      selectorOptions: {
        suppressErrors: false,
        injectContainerState: false
      }
    }),
    ... ],
  providers: [ ... ],
  entryComponents: [ ... ],
  bootstrap: [ ... ]
})
export class AppModule {}
```

NGXS - užitočné pluginy

□ Logger plugin

- ▣ Na logovanie všetkých akcií a zmien stavov
- ▣ `npm install @ngxs/logger-plugin`
- ▣ v `@NGModule` importujeme po `NgxsModule` (vždy ako posledný plugin):
 - ▣ `NgxsLoggerPluginModule.forRoot()`

□ Devtools plugin

- ▣ `npm install @ngxs/devtools-plugin --save-dev`
- ▣ na použitie „Redux Devtools extension“ v Chrome
- ▣ importujeme po `NgxsModule`
 - ▣ `NgxsReduxDevtoolsPluginModule.forRoot()`

NGXS - akcie

- akcia popisuje, kde sa stala udalosť a čo treba spraviť
- akcia je trieda s premennou **type** v tvare
 - ▣ static readonly **type** = '[zdroj] čo_spraviť'
 - [User API] UserSaved, [Product Page] AddItemToCart, [User Details Page] PasswordChanged, ...
 - ▣ kvôli prehľadnosti by akcia s rovnakým typom nemala vznikáť na dvoch miestach v kóde
- inštancia akcie môže mať dodatočné dáta
 - ▣ dodáme cez konštruktor

```
export class Login {  
  static readonly type = '[LoginPage] login';  
  constructor(public auth: Auth) { }  
}
```


NGXS – vyvolanie (dispatch) udalosti

- necháme inject-núť `store:Store` z `'@ngxs/store'`
 - ▣ v konštruktore komponentu, služby, ...
- `this.store.dispatch(new Login({name: 'jano', password: 'pekny'}));`
- môžeme zaslať aj viac akcií naraz v poli
 - ▣ `this.store.dispatch([new AddUser(user1), new AddUser(user2)]);`
- môžeme reagovať, keď sa akcia vykoná
 - ▣ `this.store.dispatch(new AddUser(user)).subscribe()`
`() => this.form.reset();`
 - `dispatch` je `Observable`, ktoré nedodá žiadne dáta
 - robiť `subscribe` pre `dispatch()` je nepovinné

NGXS – stav modelu

- aplikácia môže mať veľa modelov
 - ▣ zoznam výrobkov, stav prihlásenia, cache objektov, ...
- pre každý model zvlášť vytvárame
 - ▣ definíciu modelu – napr. GroupsModel
 - interface definujúci aké dáta v modeli budeme mať
 - ▣ stavovú triedu
 - dekorovanú s `@State<interface_modelu>()`
 - obsahuje metódy reagujúce na jednotlivé akcie
 - tie sú dekorované s `@Action(akcia_na_ktorú_reaguje)`
 - niekedy sa volajú reducers
 - obsahuje aj metódy, zvané selektory, na získanie vybranej časti aktuálneho stavu modelu

NGXS – stavová trieda - príklad

```
import { State, Action, StateContext } from '@ngxs/store';
import { Auth } from 'src/app/auth';

export class Login {
  static readonly type = '[Login Page] Login';
  constructor(public auth: Auth) {}
}

export interface AuthStateModel {
  username: string;
  token: string;
}

@State<AuthStateModel> ({
  name: 'auth',
  defaults: { username: null, token: null }
})
export class AuthState {
  @Action(Login)
  login(ctx: StateContext<AuthStateModel>, action: Login) {
    // obsluha udalosti Login
  }
  ...
}
```

akcia „Login“

model = čo si pamätáme
v stave

unikátne meno stavu v
úložisku

úvodná hodnota modelu

NGXS – reakcie (reducers)

- funkcie v stavovej triede s `@Action(trieda_akcie)`
- typicky menia stav modelu
 - ▣ VŽDY z immutable stavu na immutable stav
- na vstupe majú
 - ▣ `ctx: StateContext<interface_modelu>`
 - `ctx.getState()` – vráti aktuálny immutable stav modelu
 - `ctx.setState(nový_stav)` – nastavuje nový immutable stav
 - alebo cez `ctx.setState((starý_stav) => nový_stav)`
 - `ctx.patchState(čiasočný_stav)`
 - nahradí starý stav za nový tak, že si vytvorí klon a v ňom nahradí všetko z čiastočného stavu – platí iba pre prvú úroveň objektu/pol'a
 - `ctx.dispatch(new Iná_akcia())` alebo `ctx.dispatch([iné_akcie])`
 - vyvolá ďalšiu akciu a vráti Observable, ktoré ak vrátime ako návratovú hodnotu, ten čo volal nás bude informovaný o ukončení až po vybavení aj tejto bezprostrednej akcie
 - ▣ objekt akcie
 - zdroj dodatočných dát akcie

NGXS - immutable stav

entita:

```
export interface Item {  
  id: number;  
  name: string;  
}
```

akcia:

```
export class AddItem {  
  static readonly type =  
    '[ItemList Page] AddItem';  
  constructor(public item: Item) {}  
}
```

model:

```
export interface CartModel {  
  items: Item[];  
  paid: boolean;  
}
```

reducer – verzia 1:

```
@Action(AddItem)  
add(ctx: StateContext<CartModel>, action: AddItem) {  
  const state = ctx.getState();  
  ctx.setState({  
    ...state,  
    items: [...state.items, action.item]  
  });  
}
```

reducer – verzia 3:

```
@Action(AddItem)  
add(ctx: StateContext<CartModel>, action: AddItem) {  
  const state = ctx.getState();  
  ctx.patchState({  
    items: [...state.items, action.item]  
  });  
}
```

reducer – verzia 2:

```
@Action(AddItem)  
add(ctx: StateContext<CartModel>, action: AddItem) {  
  ctx.setState(state => ({  
    ...state,  
    items: [...state.items, action.item]  
  }));  
}
```

reducer – verzia 4:

```
@Action(AddItem)  
add(ctx: StateContext<CartModel>, action: AddItem) {  
  ctx.setState(  
    patch({ items: append([action.item]) })  
  );  
}
```

NGXS - stavové operátory

□ objekty

- **patch**(čiasočný_objekt) – nahradí vlastnosť objektu

- `ctx.setState(patch({ paid: true }))`

□ polia

- **append**([h1, h2]) – pridá na koniec poľa hodnoty h1 a h2
- **insertItem**(obj, 3) – pridá do poľa obj na pozíciu 3, zvyšok posunie doprava
- **updateItem**(el => el.id = 4, obj) – zmení element poľa, ktorého id=4, za obj
 - **updateItem**(2, obj) – zmení tretí element poľa za obj
 - **updateItem**(2, oldValue => newValue) - zmení tretí element poľa za newValue
- **removeItem**(el => el.id=4) – vymaže z poľa element s id = 4
 - **removeItem**(2) – vymaže z poľa tretí prvok

□ hocičo

- **iif**(pole => pole === null, [newObj], pole => [...pole, newObj])
 - ak platí predikát vykoná sa druhý parameter, inak tretí
 - v tomto prípade, ak je hodnota poľa null vráti sa pole s jedným prvkom, inak sa vráti pole s pridaným prvkom
 - ekvivalentne: **iif**(pole => pole === null, [newObj], **append**([newObj]))

NGXS - získanie aktuálneho stavu

- Snapshot je na aktívne získanie súčasného stavu
- `this.userName = store.select`
`.selectSnapshot(state => state.auth.username);`
 - ▣ `auth` je meno definované nad stavovou triedou v `@State<AuthStateModel> ({ name: 'auth', ...})`
- vhodné napr. v strážcoch routra

NGXS - @Select

- Ak chceme v komponente **sledovať** stav, použijeme @Select na získanie Observable, ktoré nás bude informovať o nových hodnotách

```
import { Select } from '@ngxs/store';
import { AuthState } from 'src/shared/items.state';
import { AuthStateModel } from 'src/shared/auth.state';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  @Select(AuthState) auth$: Observable<AuthStateModel>;
  ...
}
```


NGXS – selektovanie časti stavu

- pridáme dekorátor pred inštančnú premennú
 - ▣ `@Select(state => state.auth.username) username$: Observable<string>;`
 - `auth` je meno definované nad stavovou triedou v `@State<AuthStateModel> ({ name: 'auth', ...})`
- nasetujeme premennú v konštruktore:

```
export class XXXComponent implements OnInit {  
  userName$: Observable<string>;  
  
  constructor(private store: Store) {  
    this.userName$ = store.select(state => state.auth.username);  
  }  
}
```

NGXS – selektovanie časti stavu

- krajšie riešenie je vytvoriť si selektor v stavovej triede

```
@State<AuthStateModel>({ ... })  
export class AuthState {  
  @Selector()  
  static userName(currentState: AuthStateModel) {  
    return currentState.username;  
  }  
  ...  
}
```

- a potom ho použiť v komponente pre inštančnú premennú
 - ▣ @Select(AuthState.userName) username\$: Observable<String>;

NGXS - spájanie selektorov

- selektor odekorovaný s `@Selector()` je spustený vždy, keď sa zmení niečo v stave jeho stavovej triedy a vygeneruje novú hodnotu do všetkých `@Select()` Observable premenných, ktoré ho využívajú
- ako parametre v `@Selector`-e môžeme uviesť **pole stavový tried**, a/alebo **d'alších selektorov**, ktoré sa majú sledovať na zmenu hodnoty
 - ▣ ak uvedieme na vstupe pole, stav jeho stavovej triedy sa nepripojí sám a treba ho dopísať, ak ho chceme sledovať
 - ▣ ak sledujeme iný selektor, vieme redukovať počet hodnôt poslaných do `@Select()` Observable premenných, lebo sa spustíme, len ak sa zmení hodnota zo sledovaného selektora

NGXS - spájanie selektorov

```
@State<AuthStateModel>({ ... })  
export class AuthState {  
  @Selector([AuthState])  
  static token(auth: AuthStateModel) {  
    return auth.token;  
  }  
  ...  
}
```

spustí sa, keď sa zmení niečo v
stave AuthState

spustí sa, keď sa niečo
zmení v stave ItemsState

spustí sa, keď sa zmení
token v AuthState alebo
items v CartState

```
@State<CartModel>({ ... })  
export class CartState {  
  @Selector()  
  static items(cart: CartModel) {  
    return cart.items;  
  }  
  
  @Selector([AuthState.token, CartState.items])  
  static validItems(token: string, items: Item[]) {  
    return token ? items : [];  
  }  
  ...  
}
```

NGXS - dynamické selektory

- ak chceme jeden selektor použiť na viac vecí
- V `@Select`-e pri odkazovaní na selektor môžeme dodať selektoru parametre, napr:
 - ▣ `@Select(CartState.items("bicycle")) bicycles$: Observable<Item[]>;`
 - ▣ `@Select(CartState.items("doll")) dolls$: Observable<Item[]>`

```
@State<CartModel>({ ... })
export class CartState {

  static items(itemType: string) {
    return createSelector([ItemsState], (state: CartModel) => {
      return state.items.filter( item => item.type === itemType);
    });
  }
  ...
}
```

NGXS - sledovanie priebehu akcií

- Keďže konkrétne akcie sú inštancie tried (napr. akcia pridania konkrétneho výrobku do košíka), vieme sledovať priebehy akcií podľa typu
 - ▣ teda napr. všetky pridania výrobkov do košíka
- Úložisko má injektovateľný prúd všetkých udalostí typu Actions, ktorý môže hocikto sledovať
- z tohto prúdu vieme vyfiltrovať, čo nás zaujíma podľa
 - ▣ triedy (typu) akcie
 - ▣ stavu spracovania akcie

NGXS - sledovanie priebehu akcií

- na fitrovanie prúdu akcií podľa stavu máme prúdové filtre:
 - ▣ ofAction(typ)
 - ▣ ofActionDispatched(typ)
 - ▣ ofActionSuccessful(typ)
 - ▣ ofActionCanceled(typ)
 - ▣ ofActionErrored(typ)
 - ▣ ofActionCompleted(typ)

```
@Component({ ... })
export class CartComponent {

  constructor(private actions$: Actions) { }

  ngOnInit() {
    this.actions$.pipe(ofActionSuccessful(ItemDelete)).subscribe(
      () => alert('Item deleted'));
  }
}
```

NGXS - routovanie

- štandardne sa v stavových triedach nedá používať router, takže ak je potrebné ako reakciu na akciu zmeniť URL, je niekoľko možností:
 - ▣ počkať si na vykonanie akcie cez subscribe na dispatch volanie a routovať v ňom
 - ▣ odchytiť akciu napr. cez ofActionSuccessful(..)
 - niekedy je ťažké vybrať, kde túto akciu odchyťovať (app komponent?, service?)
 - ▣ router plugin

NGXS – router plugin

- inštalujeme
 - ▣ `npm install @ngxs/router-plugin`
- dopíšeme do imports v `@NgModule`
 - ▣ `NgxsRouterPluginModule.forRoot()`
- v úložisku pribudne stav **'router'** s hodnotou typu `RouterStateSnapshot`
- generuje akcie do prúdu akcií:
 - ▣ `RouterNavigation`
 - ▣ `RouterCancel`
 - ▣ `RouterError`
 - ▣ `RouterDataResolved`
- môžeme vyvolať akciu **Navigate** na zmenu URL

NGXS – router plugin

□ navigácia z úložiska:

```
@Action(Login)
login(ctx: StateContext<AuthStateModel>, { auth }: Login) {
  return this.userService.login(auth).pipe(
    tap(token => {
      ctx.setState({ username: auth.name, token });
    }),
    mergeMap(() => ctx.dispatch(new Navigate(['/'])))
  );
}
```

NGXS - storage plugin

- umožňuje ukladať stav do localStorage, sessionStorage alebo vlastného
- `npm install @ngxs/storage-plugin --save`
- importujeme v `@NgModule`
 - ▣ `NgxsStoragePluginModule.forRoot({
 key: "auth.token"`
 })
 - defaultne ukladá do localStorage, ak chceme iný, použijeme okrem key aj
 - `storage: StorageOption.SessionStorage`
 - key môže byť aj pole klúčov

NGXS - zrušenie predchádzajúcej akcie

- Keď napr. používateľ klikne viackrát a vygeneruje tak znova akciu, keď ešte reakcia na ňu nedobehla, môžeme starú reakciu zrušiť a začať novú

```
@Action(Login, { cancelUncompleted: true })
login(ctx: StateContext<AuthStateModel>, { auth }: Login) {
  this.userService.login(auth).pipe(
    tap(token => {
      ctx.setState({
        username: auth.name,
        token
      });
    })
  );
}
```

NGXS – životný cyklus stavovej triedy

- Podobne ako komponenty, ak stavové triedy môžu mať metódy reagujúce na ich životný cyklus
 - ▣ `ngxsOnInit()` – spustí sa po nastavení iníciačného stavu
 - ▣ `ngxsAfterBootstrap()` – celá aplikácia bola zrenderovaná

NGXS – úložisko pre vlastné moduly

- Ak chceme vo vlastnom module používať súkromné úložisko, je potrebné ho v `@NgModule` importovať až po hlavnom úložisku

```
// feature.module.ts
@NgModule({
  imports: [
    NgxsModule.forFeature([FeatureState])
  ]
})
export class FeatureModule {}
```

```
// app.module.ts
@NgModule({
  imports: [
    NgxsModule.forRoot([]),
    FeatureModule,
  ]
})
export class AppModule {}
```