

# ANGULAR



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

# Tradičné webové aplikácie

- Web = HTML + CSS + JS
- Na serveri čaká aplikácia, ktorá ich generuje
- Archaický prístup
  - ▣ Skriptovací jazyk zlepúje HTML súbor a posiela ho prehliadaču
  - ▣ Zdroják = kúsky HTML na striedačku s kusmi kódu, ktorý HTML dotvára
  - ▣ JS na webe slúži iba na spríjemnenie práce s webom – animácie, kontrola vstupov vo formulári

# Moderné server-side webové aplikácie

- Stále platí: server generuje HTML + CSS + JS a prehliadač ich len interpretuje
- MVC na serveri
  - ▣ HTML šablóny so špeciálnymi tagmi/atribútmi, ktoré sa odkazujú na komponenty aplikácie
  - ▣ Komponent aplikácie na základe svojho modelu doplní/nahradí príslušnú časť šablóny
  - ▣ Výsledok = šablóna + reprezentácia komponentov sa posiela zo servera ako výsledné HTML + CSS + JS
- Keď sa zmení model, dotiahne sa zo servera iba tá časť HTML, ktorá predstavuje daný komponent – AJAX volania
- V mnohých jazykoch: Java, C#, PHP, JavaScript,...
  - ▣ Java: JavaServer Faces, Apache Wicket, Vaadin,...

# Moderné client-side webové aplikácie

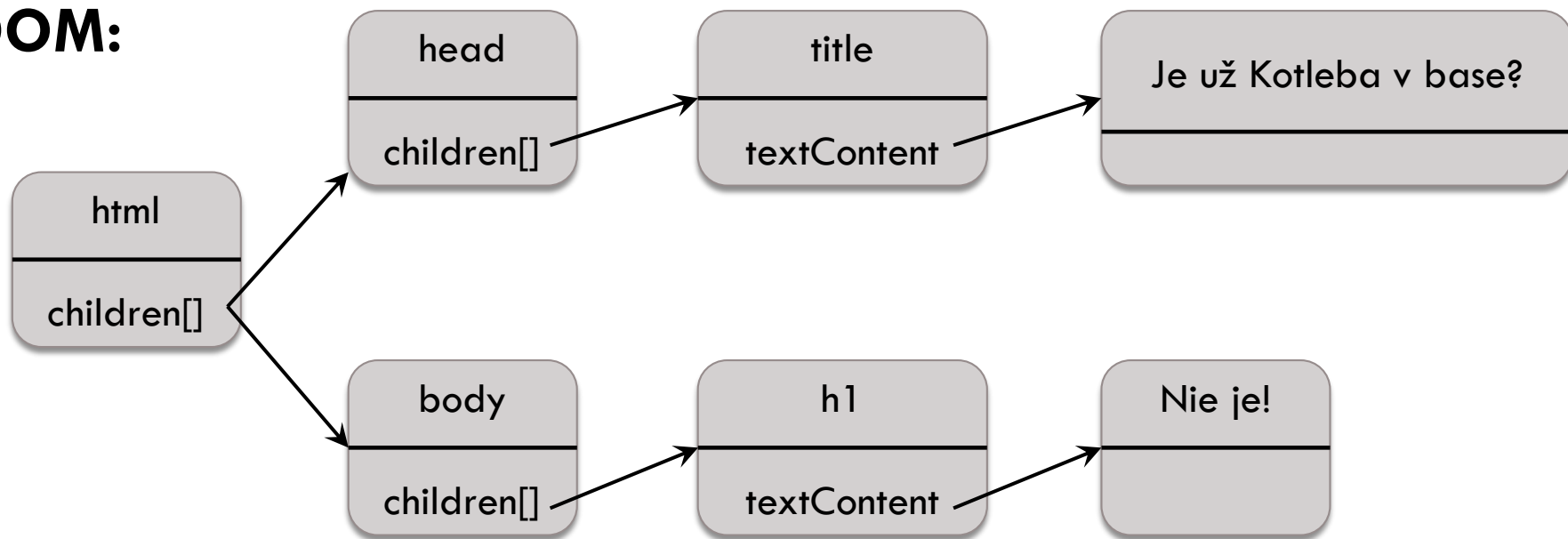
- Čistý JavaScript
- a.k.a. single-page applications, fat (thick) client
- Server poskytuje/prijíma entity – REST volania
  - ▣ perzistentná a business vrstva + REST service
  - ▣ ľubovoľný jazyk (Java, PHP, JavaScript,...)
- MVC v prehliadači
  - ▣ aplikácia modifikuje priamo DOM model zobrazovanej stránky
  - ▣ komponenty predstavujú podstromy DOM modelu
  - ▣ modely komponentov typicky predstavujú entity poskytované REST serverom
- Angular, React, Vue.js, Ember, Meteor, ExtJS, Aurelia,..

# HTML:

```
<html>
  <head>
    <title>Je už Kotleba v base?</title>
  </head>
  <body>
    <h1>Nie je!</h1>
  </body>
</html>
```

DOM model tvoria JS objekty typu  
**Node**  
a okrem textových uzlov aj typu  
**Element, HTMLElement, ...**

# DOM:



# Angular (od verzie 2)

- Final release verzie 2: 14.9.2016, aktuálne verzia 13
- Využíva jazyk TypeScript
  - ▣ Typovaná nadstavba JavaScriptu (od ECMAScript 6)
    - už existujú triedy (syntaktický cukor)
    - moduly
    - lambdy
    - ...
  - ▣ dodané anotácie, generiká
- framework = dopĺňame komponenty do štartovacej aplikácie

# Vývojové prostredie

- Mnoho mágie na pozadí
  - ▣ Typescript sa kompiluje do javascriptu, ktorému rozumie webový prehliadač
  - ▣ NodeJS server poskytuje skompilované súbory prehliadaču
  - ▣ Prehliadačom sa napojíme na NodeJS server na `http://localhost:4200/`
  - ▣ Prehliadač si stiahne súbory a spustí našu aplikáciu
    - Chová sa tak, ako sa bude chovať, keď sa nasadí naostro

# Rozbehávame vývojové prostredie



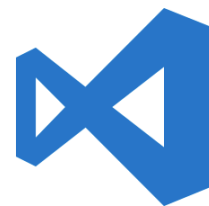
- Nainštalujeme nodejs
  - <https://nodejs.org/>
  - s ním dostaneme aj balíčkováč npm



- Nainštalujeme TypeScript
  - `npm install -g typescript`



- IDE (od Microsoftu):
  - Visual Studio Code: <http://code.visualstudio.com/>
- Nainštalujeme si zostavovač Angular projektu
  - `npm install -g @angular/cli`



- Vytvoríme si kostru projektu a fixneme balíčky
  - `ng new PROJECT_NAME`
  - `cd PROJECT_NAME`



# Import Bootstrap frameworku

- Čo je  Bootstrap ?
  - ▣ de-facto štandard na pekné štýlovanie HTML stránok (CSS + JS)
  - ▣ <http://getbootstrap.com/>
  - ▣ <http://www.w3schools.com/bootstrap/>
- Nainštalujeme ho do projektu
  - ▣ `npm install bootstrap`

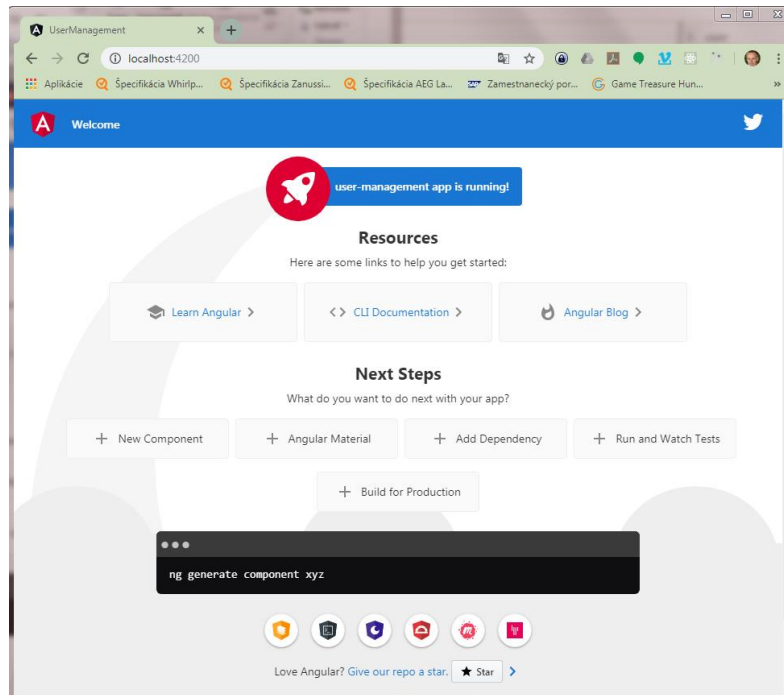
# Import Bootstrap frameworku

- Doplníme do angular.json

```
"apps": [{  
  ...  
  "styles": [  
    "node_modules/bootstrap/dist/css/bootstrap.min.css",  
    "node_modules/bootstrap-icons/font/bootstrap-icons.css",  
    "styles.css" ],  
  "scripts": [  
    "node_modules/bootstrap/dist/js/bootstrap.bundle.min.js"  
  ],  
  ...  
}]
```

# Spúšťame projekt

- Vojdeme do koreňa projektu a spustíme NodeJS server:
  - `ng serve`
- V prehliadači zadáme URL `http://localhost:4200/`



# Čo nám to vzniklo?

## □ node\_modules

□ ...

kopa modulov/knižníc, ktoré môžeme využiť

## □ src

### □ app

■ **app.component.css**

■ **app.component.html**

■ app.component.spec.ts

■ **app.component.ts**

■ **app.module.ts**

tu sa budú nachádzať naše veci

HTML šablóna koreňového komponentu

trieda koreňového komponentu

Koreňový modul  
modul= obal pre množinu komponentov,  
služieb, definuje koreňový komponent,...

### □ **index.html**

### □ main.ts

□ ...

Hlavná stránka, ktorá sa v tele odkazuje  
na koreňový komponent  
(telo nemáme, max. hlavičku)

# Letmo o OOP v Typescripte

```
class Student {  
  meno:string;  
  vek: number;  
  constructor(name:string, vek:number = 19) {  
    this.name = name;  
    this.vek = vek;  
  }  
  povedzMenoAVek(musis:boolean): string {  
    if (musis) return meno+ " " + vek;  
    else return "nepoviem";  
  }  
}
```

```
let jano= new Student("Jano");  
let janoPovedal = jano.povedzMenoAVek(true); // "Jano 19"
```

# Kód vo viacerých súboroch

```
// student.ts
```

```
export class Student {
```

```
  ...
```

```
}
```

```
export class SuperStudent extends Student {
```

```
  ...
```

```
}
```

```
// tabulka.ts
```

```
import {Student, SuperStudent} from './student';
```

```
let jano = new Student("Jano", 25);
```

```
let brunhilda = new SuperStudent("Brunhilda");
```

# Vloženie komponentu do stránky

## src/index.html

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>názov aplikácie</title>
  <base href="/">
  ...
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

## src/app/app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title:string = 'app';
}
```

## src/app/app.component.html

```
<h1>
  Welcome to {{title}}!
</h1>
```

zobrazenie  
inštancnej  
premennej

# Vložme vlastný komponent do hlavného

- Ideme do príkazového riadku a v adresári `src/app` spustíme
  - `ng g component users`
- Komponent sa importne do `src/app-module.ts`
- Vznikú 4 súbory komponentu
  - Ako selector v `src/app/users/users.component.ts` sa nastaví **app-users**
- Ak ho chceme vidieť v inom komponente, pridáme do jeho šablóny
  - `<app-users>info o používateľoch</ app-users>`



# Zobrazenie jednotlivých hodnôt

## app/users/users-component.ts

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

@Component(...)
export class UsersComponent implements OnInit {
  title: string = "Zoznam používateľov";
  users: string[] = ["Janko", "Marienka"];
}
```

## app/users/users-component.html

```
<h2>{{title}}</h2>
<ul>
  <li>{{users[0]}}</li>
  <li>{{users[1]}}</li>
</ul>
```

# Zobrazenie zoznamu cez direktívu \*ngFor

## app/users/users-component.ts

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

@Component(...)
export class UsersComponent implements OnInit {
  title: string = "Zoznam používateľov";
  users: string[] = ["Janko", "Marienka"];
}
```

## app/users/users-component.html

```
<h2>{{title}}</h2>
<ul>
  <li *ngFor="let user of users">{{user}}</li>
</ul>
```

# Radšej pracujme s entitami

```
// app/user.ts
export class User {
  constructor(
    public name: string,
    public email: string,
    public id?: number,
    public lastLogin?: Date,
    public password: string = "
  ) { }
}
```

# Vyrobme si triedu entity User

```
// app/user.ts
export class User {
  constructor(
    public name: string,
    public email: string,
    public id?: number,
    public lastLogin?: Date,
    public password: string = "
  ) { }
}
```

triedu budeme vedieť použiť  
v iných súboroch

# Vyrobme si triedu entity User

```
// app/user.ts
export class User {
  constructor(
    public name: string,
    public email: string,
    public id?: number,
    public lastLogin?: Date,
    public password: string = ""
  ) {}
}
```

nie všetky parametre sa pri  
volaní konštruktora musia  
zadat',  
id a lastLogin sú defaultne  
**undefined**,  
password je defaultne  
**prázdny string**

# Vyrobme si triedu entity User

```
// app/user.ts
export class User {
  constructor(
    public name: string,
    public email: string,
    public id?: number,
    public lastLogin?: Date,
    public password: string = ""
  ) {}
}
```

z parametrov konštruktora sa stanú verejne prístupné inštančné premenné

```
jano = new User("jano", "jano@jano.sk", 2);
console.log(jano.email);
```

# Pre neúnavných pisateľov

```
export class User {  
    private _name: string;  
  
    set name(newName: string) {  
        this._name = newName;  
    }  
    get name(): string {  
        return this._name;  
    }  
}
```

```
jano = new User();  
jano.name = "Jano";  
console.log(jano.name);
```

# Iterujeme entity

## app/users/users-component.ts

```
export class UsersComponent implements OnInit {  
  ...  
  users: User[] = [new User("Janko","jano@jano.sk"),  
                    new User("Marienka","marienka@jano.sk")];  
}
```

## app/users/users-component.html

```
...  
<ul>  
  <li *ngFor="let user of users">{{user.name}}, {{user.email}}</li>  
</ul>
```

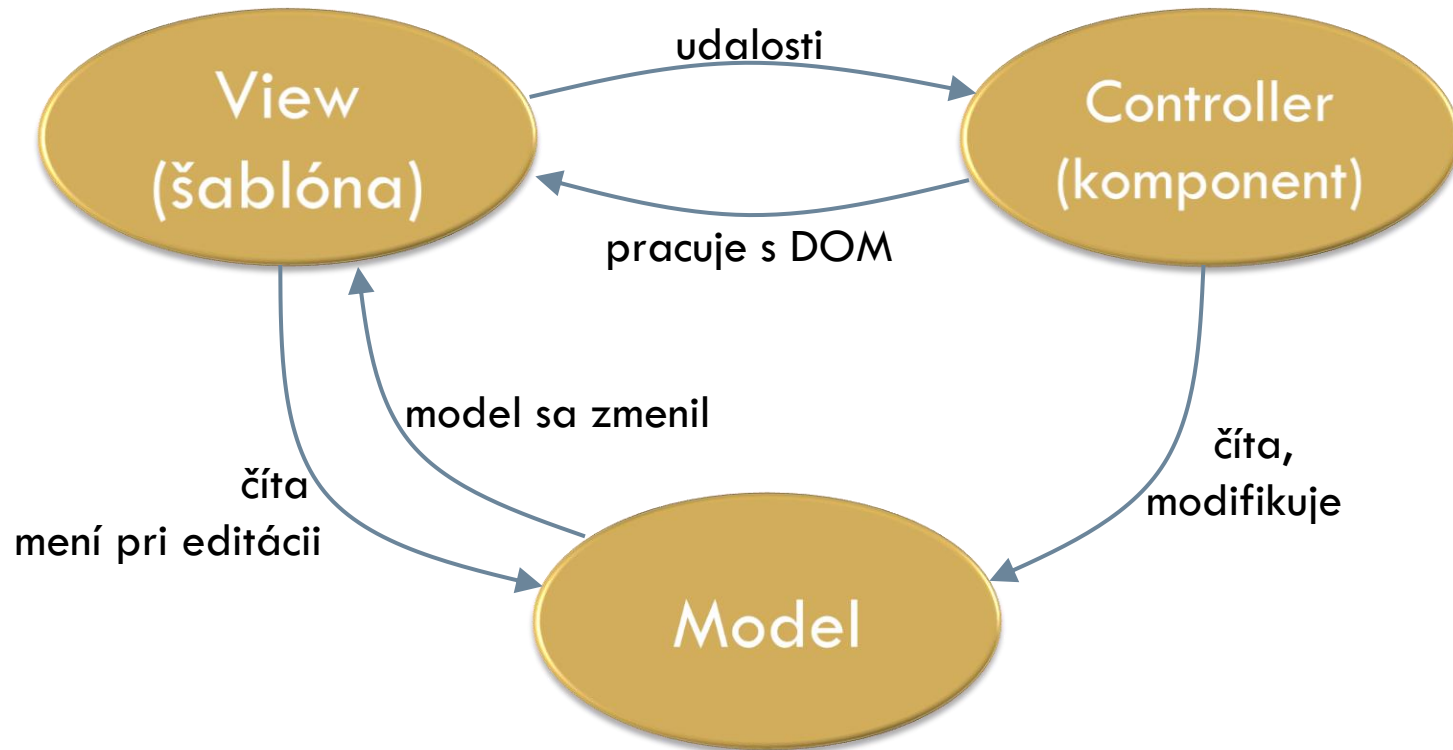


# ...alebo do tabul'ky

```
<table class="table table-hover">
  <thead>
    <tr><th>id</th><th>meno</th><th>e-mail</th></tr>
  </thead>
  <tbody>
    <tr *ngFor="let user of users" >
      <td>{{user.id}}</td>
      <td>{{user.name}}</td>
      <td>{{user.email}}</td>
    </tr>
  </tbody>
</table>
```

# Model – View – Controller (MVC)

- Odchytáva používateľské akcie
- Prekresľuje GUI, keď sa dozvie zmenu modelu
- Spracováva vstup od používateľa
- Mení alebo vymieňa model



- Uchováva zobrazovaný obsah
- Poskytuje dáta pre View, keď ich potrebuje

# Odchytenie udalosti click

## app/users/users-component.html (čast')

```
<tr *ngFor="let user of users" (click)="selectUser(user)">
  <td>{{user.id}}</td>
  <td>{{user.name}}</td>
  <td>{{user.email}}</td>
</tr>
```

## app/users/users-component.ts

```
export class UsersComponent implements OnInit {
  ...
  selectedUser: User | undefined;

  selectUser(user: User) {
    this.selectedUser = user;
  }
}
```

# Podmienené časti šablóny

## **app/users/users-component.html** (časť)

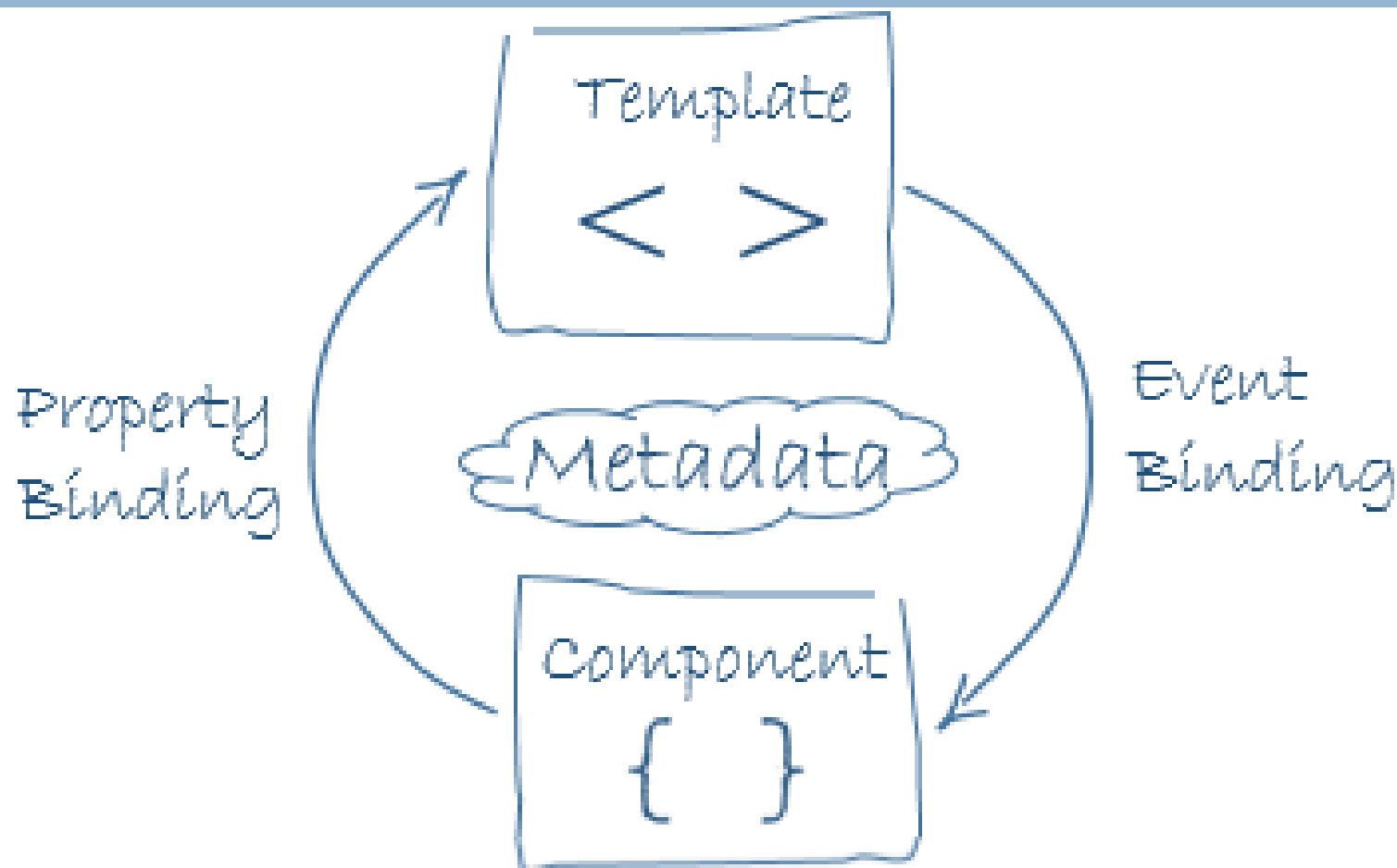
```
<tr *ngFor="let user of users" (click)="selectUser(user)">
  <td>{{user.id}}</td>
  <td>{{user.name}}</td>
  <td>{{user.email}}</td>
</tr>
<p *ngIf="selectedUser">Vybratý používateľ: {{selectedUser.name}} </p>
```

## **app/users/users-component.ts**

```
export class UsersComponent implements OnInit {
  ...
  selectedUser: User | undefined;

  selectUser(user: User) {
    this.selectedUser = user;
  }
}
```

# Architektúra komponentov Angularu



# Služby (Services)

- Časom môžeme mať viac komponentov, ktoré potrebujú používateľov
- Používateľov nemá spravovať komponent, ale perzistentná vrstva na serveri
- Náš cieľ - vytvoriť službu starajúcu sa o pole users
  - ▣ bude komunikovať s REST serverom
  - ▣ a sprostredkovať tak pre komponenty CRUD operácie nad používateľmi na serveri
- Najprv si spravíme komunikáciu služby s komponentmi
  - ▣ Služba bude zatiaľ poskytovať len svoje lokálne pole používateľov

# Vytvorenie služby

- Prejdeme do adresára, kde chceme mať službu, napr. `src/app/services`
- Spustíme príkaz
  - ▣ `ng g service users`
- Vzniknú 2 súbory
  - ▣ `src/app/services/users.service.spec.ts`
  - ▣ **`src/app/services/users.service.ts`**
    - Vytvoríme getter vracajúci pole používateľov

# Komponent potrebuje službu - injeckneme

## app/users/users-component.ts

```
import { UsersService } from '../services/users.service';  
...  
export class UsersComponent implements OnInit {  
  
  constructor(private usersService: UsersService) {}  
}
```

Čo sa injektuje cez konštruktory, musí byť zaregistrované medzi providermi (pre Angular <=5)

## app/app.module.ts

## app/users/users-service.ts

```
@Injectable({  
  providedIn: 'root'  
})  
export class UsersService {  
  
}
```

priama registrácia od Angular 6

```
import { UsersService } from  
'./users.service';  
@NgModule({  
  ...  
  providers: [UsersService],  
  ...  
})  
export class AppModule { }
```



# Komponent potrebuje service

**app/users/users-component.ts**

```
export class UsersComponent implements OnInit {  
  constructor(private userService: UsersService) {}  
  
  ngOnInit() {  
    this.updateUsers();  
  }  
  
  updateUsers() {  
    this.users = this.userService getUsers();  
  }  
}
```

Čo môže trvať dlho\*  
nerobíme v konštruktoze!

\* napr. komunikácia so  
serverom

Túto metódu  
musíme v službe  
vytvoriť

# Príprava na dlhé čakanie na dáta

## □ Synchronné volanie:

- Component si vypýta dáta od servisu a čaká
- Service si vypýta dáta zo servera a čaká
- Pokiaľ sa čaká, žiaden JavaScript nefunguje (udalosti používateľa – kliky, editácia,...)

## □ Asynchronné volanie

- Component si vypýta dáta od servisu a zaeviduje funkciu, ktorá sa má spustiť, keď dáta dôjdu
- Service si vypýta dáta zo servera a zaeviduje funkciu, ktorá sa má spustiť, keď dáta dôjdu
- Pokiaľ sa čaká na dáta, všetko funguje

# Observable

- ❑ Trieda predstavujúca obal pre premennú, ktorá sa môže zmeniť
- ❑ Vieme zaregistrovať metódu, ktorá sa má spustiť, keď sa zmení, cez metódu `subscribe( metóda )`
- ❑ Metóda sa zvykne vkladať ako lambda výraz
- ❑ Moderný spôsob robenia asynchrónnych volaní
- ❑ Zmeňme metódu v službe:
  - ❑ Operátor „of“ vyrobí nový objekt typu `Observable`
  - ❑ Musíme ho importovať

```
import { Observable, of } from 'rxjs';
```

```
getUsers(): Observable<User[]> {  
    return of(this.users);  
}
```

Zatiaľ vyrobíme jednoduchý `Observable` obal zo statickej hodnoty už naplnenej premennej

# Zaregistrovanie poslucháča v komponente

## app/users/users-service.ts

```
getUsers(): Observable<User[]> {  
  return of(this.users);  
}
```

## app/users/users-component.ts

```
updateUsers() {  
  this.userService.getUsers().subscribe(users => this.users = users);  
}
```

Ked' sa hodnota Observable zmení, spustí funkciu (v tomto prípade lambdu)

# REST server

- Použijeme predpripravný REST server,
  - ▣ beží na učiteľskom počítači s IP adresou xxx.xxx.xxx.xxx
  - ▣ stiahneme si súbor films-server.jar
    - `java -jar films-server.jar`
- Keď ho spustím počúva na adrese `http://xxx.xxx.xxx.xxx:8080/`
  - ▣ Vyskúšame cez Postman-a `http://xxx.xxx.xxx.xxx:8080/users`
- Naša Angular aplikácia je prístupná cez NodeJS server na `http://localhost:4200/`

# Pripravíme si Angular

## app.module.ts

```
import { HttpClientModule } from '@angular/common/http';

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

# getUsers() cez AJAX volanie

## app/users/users-service.ts

```
import { HttpClient } from '@angular/common/http';  
import { Observable } from 'rxjs/Observable';
```

```
...
```

```
  constructor(private http: HttpClient) { }
```

Injektujeme inštanciu HttpClient

```
  private restServerUrl: string = "http://xxx.xxx.xxx.xxx:8080/users";
```

```
  getUsers(): Observable<User[]> {  
    return this.http.get<User[]>(this.restServerUrl);  
  }
```

```
...
```

Ak JSON, ktorý príde, má rovnakú štruktúru, ako naša trieda User, mapovanie sa spraví samé na všetkých zhodných inštančných premenných

# Ručné spracovanie response, ak je potrebné

## app/users/users-service.ts

```
import { map } from 'rxjs/operators';
...
getUsers(): Observable<User[]> {
  return this.http.get(this.restServerUrl)
    .pipe(map(response => this.handleGetUsersResponse(response)));
}

private handleGetUsersResponse(jsonUsers):User[] {
  let remoteUsers:User[] = [];
  for (let jsonUser of jsonUsers) {
    remoteUsers.push(new User(jsonUser.name,jsonUser.email,jsonUser.id));
  }
  return remoteUsers;
}
```

Žiadne <User[]> za get



# Chyby pri Observable

- Ak sa komunikácia nepodarí, funkcia vložená cez subscribe sa nespustí, ale vyletí chyba do konzoly, a všetko za tým sa nevykoná
- Ak chceme túto chybu odchytiť, môžeme v subscribe vložiť dve funkcie zabalené v objekte – jedna na spracovanie dát a druhá na spracovanie chyby

## app/users/users-component.ts

```
updateUsers() {  
  this.userService getUsers().subscribe( {  
    next: users => this.users = users,  
    error: error => console.log('chyba komunikácie: ' + JSON.stringify(error))  
  } );  
}
```

# Zobrazme spätnú väzbu používateľovi

- Na zobrazenie môžeme použiť Bootstrap triedu alert
- Zobrazíme takýto alert (pozitívny alebo negatívny) ako súčasť stránky,
- Keď Observable dodá dáta - zobrazíme zmysluplnú hlášku používateľovi

# Routing

- Vytvoríme si komponent na prihlásenie
- Chceli by sme ho vidieť **namiesto** zoznamu používateľov
- Cez routing vieme mapovať rôzne koncovky našej URL adresy na rôzne komponenty
  - ▣ <http://localhost:4200/users>
    - Stránka so zoznamom používateľov
  - ▣ <http://localhost:4200/login>
    - Stránka s prihlasovacím formulárom

# Pripravíme si routing

- `cd src/app`
- `ng g component login`
  - ▣ vytvorí komponent, kde budeme vyrábať prihlasovací formulár
- `ng g module app-routing --flat --module=app`
  - ▣ vytvorí súbor `app-routing.module.ts` v adresári `src/app`
  - ▣ `--module=app` ho zaregistruje v `AppModule`

# Nastavenie routovania v app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { UsersComponent } from './users/users.component';
import { LoginComponent } from './login/login.component';

const routes:Routes = [
    { path: 'users', component: UsersComponent },
    { path: 'login', component: LoginComponent }
];

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

# Nastavenie routovania v app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes } from '@angular/router';
import { UsersComponent } from './users';
import { LoginComponent } from './login';
```

koniec URL adresy  
http://localhost/**users**

Komponent, ktorý sa  
má natiahnuť

```
const routes: Routes = [
  { path: 'users', component: UsersComponent },
  { path: 'login', component: LoginComponent }
];
```

Metóda forRoot()  
vykoná navigáciu na  
správny komponent  
na základe aktuálnej  
URL

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

Aby tento modul bol  
videný v app.module

# Nastavíme AppComponent

```
<div style="text-align:center">  
    <h1>Správa používateľov</h1>  
</div>  
<app-users>info o používateľoch</app-users>  
<router-outlet></router-outlet>
```

## □ Vyskúšame

- ▣ <http://localhost:4200/users>
- ▣ <http://localhost:4200/login>
- ▣ <http://localhost:4200/>

# Redirect z / na /users

- Pridáme ďalšiu cestu do app-routing.module.ts

```
{ path: '', redirectTo: '/users', pathMatch: 'full' }
```

- kedykoľvek sa znavigujeme na `http://localhost:4200/` tak nás presmeruje na `http://localhost:4200/users`
- `pathMatch`
  - ▣ `full` – za `path ''` v URL už nič ďalšie nie je
  - ▣ `prefix` – stačí ak sa prefix URL a path zhoduje, za ním v URL ešte niečo môže byť



# Page not found

- keď sa používateľ nanaviguje na neexistujúcu URL, chceme mu zobrazit' vlastnú stránku s chybou 404

```
{ path: '**', component: PageNotFoundComponent }
```

- Na poradí pravidiel záleží: použije sa prvé pravidlo, ktorého path sa zhoduje s URL
- Pravidlo s path= '\*\*', ktoré sa zhoduje s každou URL preto píšeme ako posledné

# REST server - metódy

Ďalší preddefinovaní  
používatelia a ich heslá :

Lucia : lucia

John : john

Andrej : andre

- GET: /users
- POST: /login
  - ▣ V tele pošleme {"name": "Peter", "password": "upjs"}
  - ▣ príde vygenerovaný token
- GET: /users/{token}
  - ▣ prídú používatelia aj s neverejnými atribútmi
- GET: /user/{id}/{token}
- GET: /bg-user/{id}/{token}
  - ▣ vnútorná reprezentácia používateľ'a (obsahuje aj heslo)
- POST: /users/{token}
  - ▣ Cez POST pošleme JSON používateľ'a, ktorého chceme uložiť
- DELETE: /user/{id}/{token}

# Prihlásenie používateľa - plán

- Spravíme si triedu Auth s premennými name a password
  - ▣ Budeme posielat' JSON objektu – konverzia sa spraví sama
- V LoginComponent si vytvoríme prihlasovací formulár s tlačidlom, ktoré iniciuje poslanie cez service komunikáciu so serverom
  - ▣ Nájdeme si na internete nejakú šablónu pre login formulár
- Použijeme HTTP metódu POST

# Model editačného komponentu

- Základný model komponentu, v ktorom budeme editovať prihlasovacie údaje je objekt typu Auth
- Môžeme si ho v komponente vyrobiť prázdneho, aby sa prvky šablóny mali s čím previazať

```
...  
export class LoginComponent implements OnInit {  
  auth:Auth = new Auth("", "");  
  ...  
}
```

# Pomocný text s obsahom modelu/Auth

## **app/login/login.component.html (časť)**

```
<div class="modal-body">  
  <p>aktuálne údaje: {{vypisAuth}}</p>  
</div>
```

## **app/login/login.component.ts**

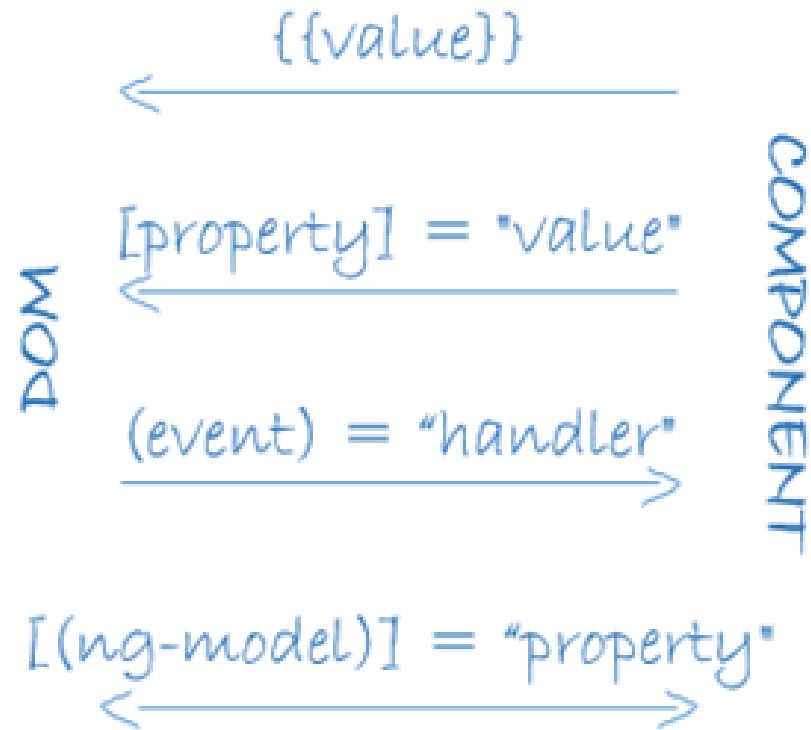
```
get vypisAuth():string {  
  return JSON.stringify(this.auth);  
}
```

# Udalosti vo formulári

- Jedna z hlavných výhod MVC v prehliadači je jednoduchá spätná väzba pri editácii formulárov
- Webový formulár nám mení model komponentu
  - ▣ ten môžeme okamžite vyhodnocovať, či je ok
  - ▣ ...a dať používateľovi spätnú väzbu

# Prepojenie DOM a modelom v komponente v Angulari

- DOM sa buduje ako kombinácia šablóny a obsahu modelu
- V DOM sa odchyťávajú udalosti používateľa, ktoré môžeme poslať komponentu
  - ▣ zavolaním metódy
  - ▣ obojsmerným prepojením s modelom cez ngModel



# Pripravíme si Angular

## app.module.ts

```
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';

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



# Udalosti

## app/login/login.component.html (časť)

```
<input type="text" (keyup)="setAuthName($event)" value="{{auth.name}}" />
```

## app/login/login.component.ts

```
setAuthName(event:any) {  
  this.auth.name = event.target.value;  
}
```

- premenná `$event` obsahuje informácie o udalosti
- na zobrazenie obsahu potrebujeme aj mapovanie z komponentu do DOM
- potrebujeme obslužnú metódu
- potrebujeme vedieť, že element `input` má atribút `value`

# Previazanie modelu komponentu a DOM

## Pred tým:

```
<input type="text" (keyup)="setAuthName($event)" value="{{auth.name}}" />
```

## Po tom:

```
<input type="text" [(ngModel)]="auth.name" name="login" />
```

Ľubovoľné unikátne meno pre každý ngModel formulára

- ngModel previaže obojstranne element formulára s premennou modelu komponentu
- nepotrebujeme obslužnú metódu
- nepotrebujeme vedieť, že aký element obsluhujeme

# Odoslanie obsahu formuára serveru

- ...heslo user-a urobíme analogicky
- odoslanie spravíme cez tlačidlo na uloženie
  - ▣ obalíme celú šablónu do elementu `<form>`
  - ▣ v komponente vytvoríme obslužnú metódu `onSubmit()`

**app/login/login.component.html** (časť)

```
<form (ngSubmit)="onSubmit()" ... >
```

```
...
```

```
<button type="submit" ... >Sign in</button>
```

```
</form>
```

# Prihlásenie používateľa - service

## app/users/users-service.ts

```
import { HttpClient } from '@angular/common/http';
import { Auth } from '../auth';

...
private restServerUrl: string = "http://localhost:8080/";

login(auth: Auth):Observable<string> {
    return this.httpClient.post(this.restServerUrl + "login", auth,
                                {responseType : 'text'});
}
```

Nepríde JSON ale  
iba string

- Vrátime tak token komponentu, ktorý bude volať subscribe na Observable z login()

# Uvažujme inak

- Token komponentu netreba (nezobrazujeme ho)
  - ▣ Môže si ho získať aj service
  - ▣ Nemusíme pri každom volaní servisných metód posilať token ako parameter
  - ▣ Viaceré komponenty môžu využiť servisné metódy s tokenom
- Zároveň však chceme poslať komponentu informáciu o úspechu prihlásenia
- Kto má volať `subscribe()` na `Observable`?
  - ▣ `subscribe()` zavolá stále komponent, lebo iniciuje spustenie komunikácie pri stlačení tlačidla
- Service môže pristúpiť do prúdu prijatých dát pred odoslaním iniciátorovi
  - ▣ Môže prijaté dáta spracovať, dokonca aj upraviť a poslať komponentu už upravené

# Pošleme true, ak sa prihlásenie podarí

```
private token: string = null;
```

```
login(auth: Auth):Observable<boolean> {  
    return this.httpClient.post(this.restServerUrl + "login",auth,  
                                {responseType : 'text'})  
    .pipe(map(token => {  
        this.token = token;  
        return true;  
    }));  
}
```

# Ak sa heslo nezhoduje, pošleme false

```
login(auth: Auth):Observable<boolean> {  
  return this.httpClient.post(...)  
    .pipe(map(...),  
      catchError(error => {  
        if (error instanceof HttpResponse &&  
            error.status == 401)  
          return of(false); // zlé heslo  
        }  
        return throwError(error); // nejaká iná chyba  
      }));  
}
```

# Dorobíme login komponent

- Ak príde **true**, môžeme zobrazit' inú stránku

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

```
this.router.navigateByUrl('/users');    // v subscribe
```

- Ak príde **false**, zobrazíme hlášku o zlom hesle
- Ak príde chyba, zobrazíme hlášku o chybe komunikácie



# Dorobíme si navigáciu

- Vytvoríme komponent obsahujúci navigáciu
  - ▣ Nájdeme na Bootstrape príklad na NavBar
- V navigácii dáme linky na všetky zatiaľ používané URL adresy

# Životnosť servisu

- Service je singleton
  - ▣ Každý komponent, ktorý si ho nechá injektovať vidí rovnakú inštanciu
- Pri zmene URL cez `<a href="">`, však dochádza k reštartu celej stránky
  - ▣ Service sa vytvára nanovo
  - ▣ Token z predchádzajúcej URL už nebude uložený
- Vieme využiť session úložisko v prehliadači
  - ▣ Funkcie v globálnom JS objekte **sessionStorage** alebo **localStorage**:
    - `setItem(kľúč, hodnota)`
    - `getItem(kľúč)`
    - `removeItem(kľúč)`
    - `clear()`
  - ▣ max 10MB pre origin (doménu) a iba stringové hodnoty

# Router pre Single page application

- Nechceme komunikovať so serverom, keď už aj tak máme natiahnuté všetky komponenty v prehliadači
  - ▣ namiesto `<a href="/users">` použijeme:
    - `<a routerLink="users">`
  - ▣ router Angularu v tomto prípade iba vymení komponenty, ale nepýta server o novú stránku
  - ▣ na presmerovanie v kóde použijeme:

```
import { Router } from '@angular/router';
...
constructor(private router: Router){}
...
goToUsers() {
  this.router.navigateByUrl("/users");
  //alebo this.router.navigate(["/users"]); - tu vieme použiť relatívnu cestu
}
}
```

# Nastavovanie class link elementom

- zvýraznenie kliknutého elementu – aktívna linka v menu

```
<a routerLink="/users" routerLinkActive="active">Heroes</a>
```

- ▣ nastavíme class="active"
- ▣ v css nastavíme iný dizajn cez selektor .active

# Login / Logout v hlavičke stránky

- Spoločná hlavička: koreňový AppComponent
- Service má komponentu povedať, keď sa stav zmení
  - ▣ AppComponent nevie **kedý** to príde a **koľko krát** to príde – je to závislé od toho, čo urobia vnorené komponenty
  - ▣ Zaregistrujeme ho na reagovanie na akékoľvek budúce zmeny
  - ▣ AppComponent pri každej novej hodnote iba prekreslí linku

# Dlhodobé Observable

- Pri vytvorení Observable si zapamätáme jeho Subscriber – objekt ktorý posiela dáta do Observable

```
private loggedUserSubscriber: Subscriber<string>;
```

```
public loggedUser():Observable<string> {  
    return new Observable<string>( subscriber => {  
        this.loggedUserSubscriber = subscriber;  
        subscriber.next(this.user);  
    });  
}
```

```
this.loggedUserSubscriber.next("Jano");
```

```
get user() {  
    return sessionStorage.getItem("user");  
}
```

# ExtendedUsersComponent

- Zapýtame si rozšírených používateľov
  - GET: `http://localhost:8080/users/{token}`
- Rozšírime triedu User o ďalšie parametre
  - Aj pole objektov typu Group
- Zaevidujeme si komponent v app-routing
- Vypíšeme tabuľku s rozšírenými používateľmi

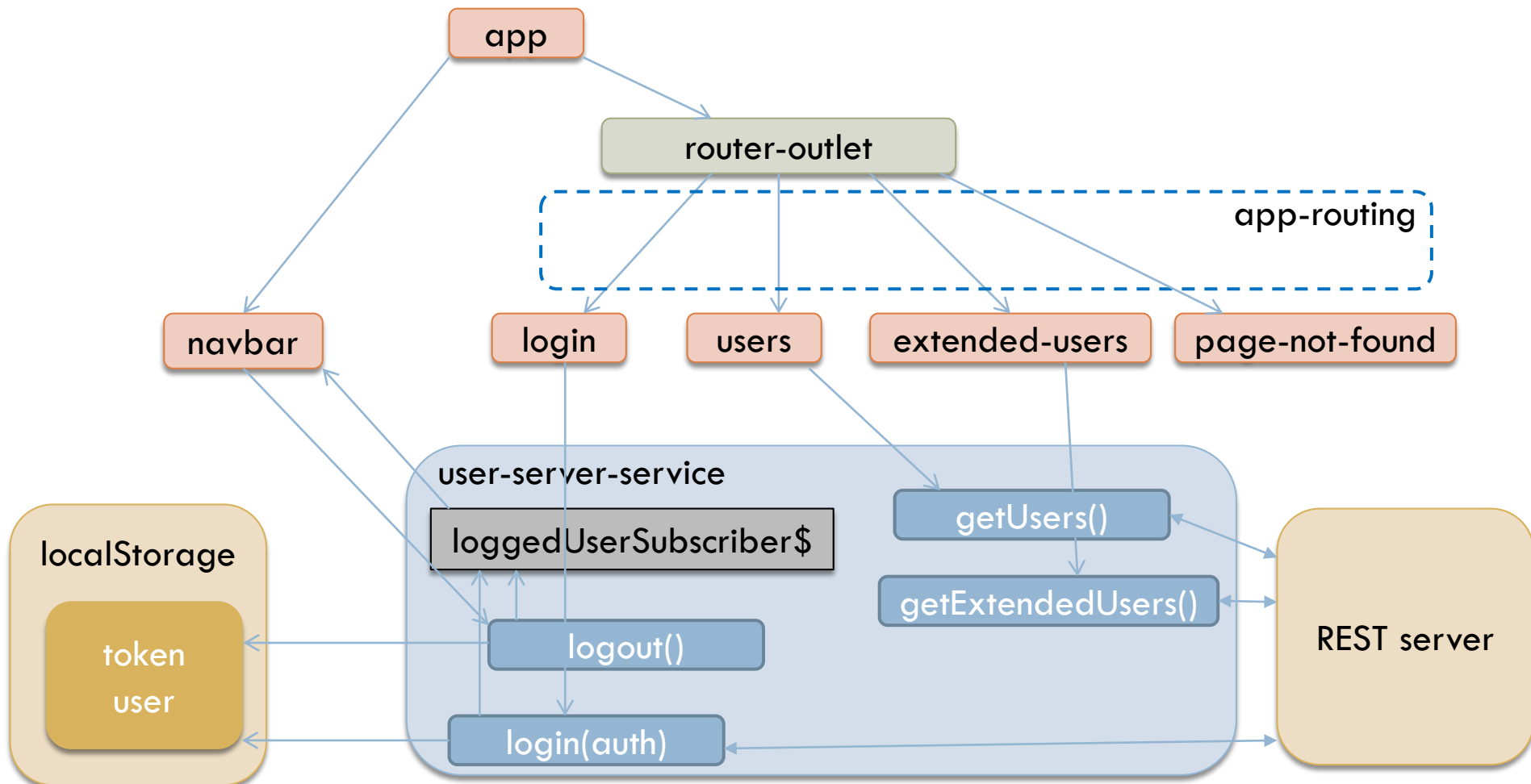
# Vlastná pipe

- Spravíme si vlastnú pipe pre výpis názvov skupín a práv z nich vyplývajúcich
  - ▣ `transform(values: Group[], property?: string): string { }`
- `ng g pipe groups-to-string`
  - ▣ importuje sa do `.module.ts`
- použijeme ju v šablóne

```
<ng-container matColumnDef="permissions">
  <th mat-header-cell *matHeaderCellDef>Permissions</th>
  <td mat-cell *matCellDef="let user">
    {{ user.groups | groupsToString: 'permissions' }}
  </td>
</ng-container>
```



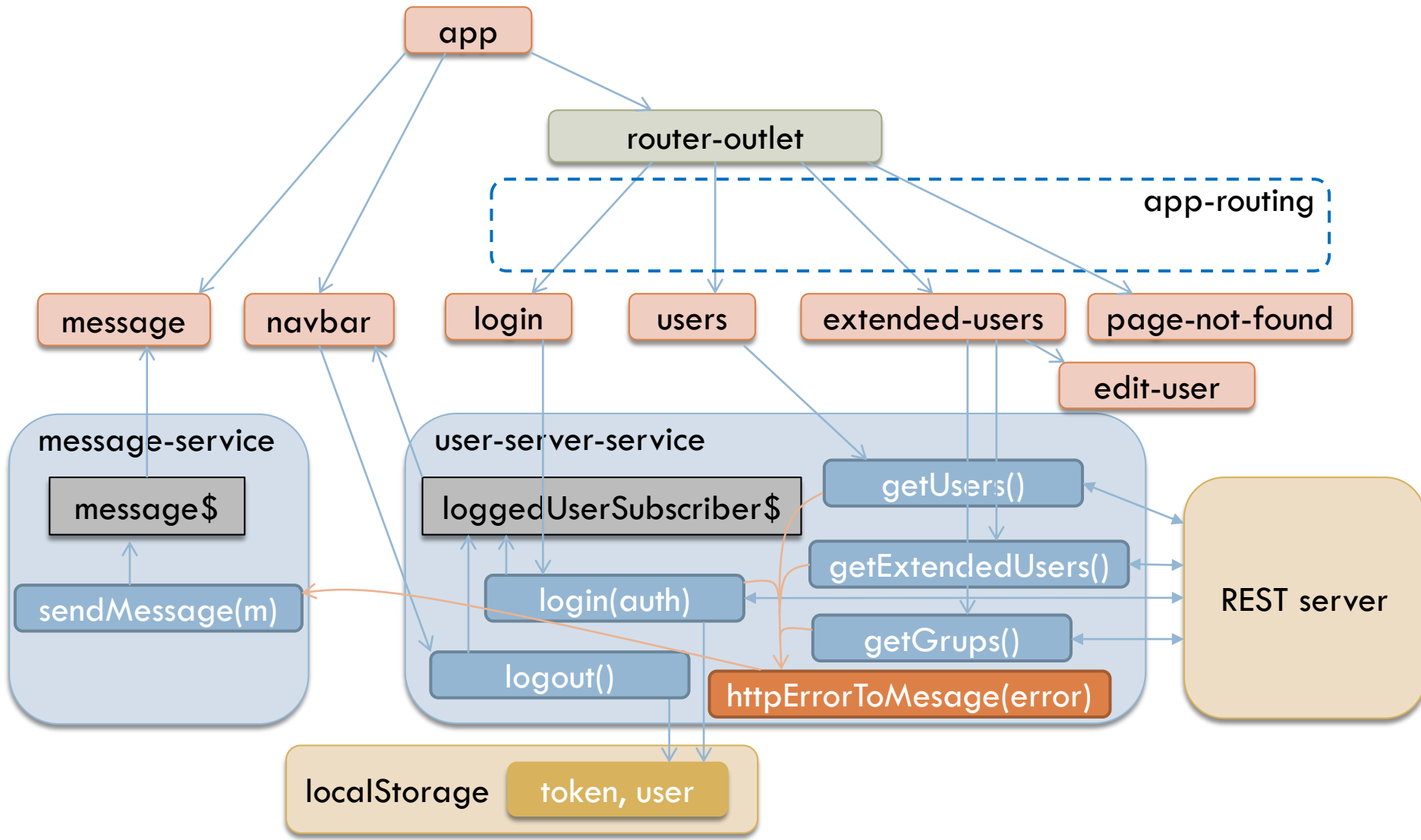
# Aktuálny stav



# Refaktor

- Chybové/úspechové hlášky by mohol vypisovať message komponent
  - vypisovať chybové, ale aj pozitívne správy
  - všetky možné chyby komunikácie
    - nedostupný server
    - zlý login alebo heslo
    - nedostatočné práva
  - prebalit' chyby do vlastného message objektu
    - poslať ho message komponentu
      - vypísať správu, nech už bude akákoľvek
    - presmerovať sa na hlavnú stránku?

# Aktuálny stav



# “subscribe” zo šablóny (Angular4+)

- Ak vieme, že chyby cez Observable neprídu
- Použijeme v šablóne rúru (pipe) `async`
- Šablóna si dokáže počkať na dáta asynchrónne
  - ▣ V šablóne pracujeme s premennou typu Observable
  - ▣ Šablóna si sama spraví subscribe a iniciuje spustenie

## users.component.html

```
private users$: Observable<User[]>;

ngOnInit() {
  this.users$ =
    this.userService.getSimpleUsers();
  ...
}
```

## users.component.ts

```
<ul>
  <li *ngFor="let user of users$ | async"
    (click)="selectUser(user)">

    {{user.name}}, {{user.email}}

  </li>
</ul>
```

# Téma: komunikácia komponentov

- **Ciel':** Vytvoríme si komponent s formulárom, v ktorom budeme môcť pridať nového používateľa
- V src/app spustíme
  - ▣ ng g component user-edit
- Ak ho chceme vidieť v komponente extended-users, vložíme do jeho šablóny tag, ktorý sa volá rovnako, ako selector v novom komponente

```
<app-user-edit></app-user-edit>
```

# Vytvoríme šablónu obsahujúcu formulár

- Spravíme si modálne okno, ktoré bude predstavovať náš nový komponent na editáciu
- Tlačidlo na jeho zobrazenie ponecháme v rodičovskom komponente `extended-users.component`
- Pozrime sa, ako sa robia modálne okná v Bootstrap a okopírujme si HTML zdrojáky



# Model editačného komponentu

- Základný model komponentu, v ktorom budeme editovať nového používateľa je objekt typu User
- Môžeme si ho v komponente vyrobiť prázdneho, aby sa prvky šablóny mali s čím previazať

...

```
export class UserEditComponent implements OnInit {  
  user: User = new User("", "");
```

...

```
}
```

# Pomocný text s obsahom modelu/user-a

## **app/user-edit/user-edit-component.html** (časť)

```
<div class="modal-body">  
  <p>aktuálny user: {{vypisUsera}}</p>  
</div>
```

## **app/user-edit/user-edit-component.ts**

```
get vypisUsera():string {  
  return JSON.stringify(this.user);  
}
```



# Editácia skupín

- Používatelia majú v sebe iba nim priradené skupiny – noví dokonca žiadne
- V systéme však sú aj ďalšie skupiny
  - ▣ Získame si ich zo servera
    - GET: /groups
- V komponente bude modelom pre vybraté skupiny zoznam objektov obaľujúcich skupinu a boolean členstva user-a v nich

```
groups: {group: Group, isMember: boolean} = [];
```

# Dynamicky vytvoríme checkboxy

```
<div *ngFor="let item of groups; let i = index">
  <label>
    <input type="checkbox" name="grchb{{i}}"
      [(ngModel)]="item.isMember">
      {{item.group.name}}
  </label>
</div>
```

# Validácia vstupu

- Nechceme, aby meno ostalo prázdne a ak áno, tak o tom informovať používateľa
- ngModel nám znova pomôže – nastavuje sám od seba pre každý formulárový element jednu 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

# Naštýľujeme si elementy podľa týchto tried

**app/user-edit/user-edit-component.css**

```
.ng-valid[required] {  
  border-left: 5px solid limegreen;  
}
```

```
.ng-invalid {  
  border-left: 5px solid darkred;  
}
```

- selektor . označuje elementy s danou triedou
- [required] znamená, že daný element musí mať aj atribút required

# Dodajme aj upozorňujúci text

app/user-edit/user-edit-component.html (časť)

Sivou sú veci z  
Bootstrap-u

Nová  
premenná

```
<div class="form-group">  
  <label>login:</label>
```

```
  <input type="text" class="form-control" [(ngModel)]="user.name"  
  name="name" required #validne="ngModel" />
```

```
  <div [hidden]="validne.valid || validne.pristine" class="alert alert-  
  danger">Meno nemôže byť prázdne.</div>
```

```
</div>
```

Nastaví elementu vlastnosť  
hidden, ak výraz napravo je  
pravdivý

# Odoslanie obsahu formulára rodičovi

- ...ostatné parametre user-a urobíme analogicky
- odoslanie spravíme cez tlačidlo na uloženie
  - ▣ obalíme celú šablónu do elementu `<form>`
  - ▣ znefunkčníme tlačidlo, ak je formulár nevalidný
  - ▣ v komponente vytvoríme obslužnú metódu `onSubmit()`

## **app/user-edit/user-edit-component.html** (časť)

```
<form (ngSubmit)="onSubmit()" #userForm="ngForm">
...
<button type="submit" class="btn btn-primary"
[disabled]="!userForm.form.valid">Ulož</button>
...
</form>
```

# Odoslanie nového user-a rodičovi

## user-edit-component.ts

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

export class UserEditComponent ...{
  @Output() eventPipe = new EventEmitter<User>();
  ...
  onSubmit() {
    this.eventPipe.emit(this.user);
  }
}
```

# Odoslanie nového user-a rodičovi

## user-edit-component.ts

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

export class UserEditComponent ...{
  @Output() eventPipe = new EventEmitter<User>();
  ...
  onSubmit() {
    this.eventPipe.emit(this.user);
  }
}
```

## extended-users-component.html

```
<app-user-info (eventPipe)="onEvent($event)"></app-user-info>
```



# Odoslanie nového user-a rodičovi

## user-edit-component.ts

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

export class UserEditComponent ...{
  @Output() eventPipe = new EventEmitter<User>();
  ...
  onSubmit() {
    this.eventPipe.emit(this.user);
  }
}
```

## extended-users-component.html

```
<app-user-info (eventPipe)="onEvent($event)"></app-user-info>
```

## extended- users-component.ts

```
onEvent(user:User) {
  this.users.push(user);
}
```

# Odoslanie nového user-a rodičovi

## user-edit-component.ts

```
import {EventEmitter, Output } from '@angular/core';
import {Modal} from 'bootstrap';
export class UserEditComponent ...{
  @Output() eventPipe = new EventEmitter<User>();
  ...
  onSubmit() {
    this.eventPipe.emit(this.user);
    const modalEl =
      document.getElementById('exampleModal');
    const modal = modalEl ? Modal.getInstance(modalEl) : null;
    if (modal) modal.hide(); // alebo 'toggle'
  }
}
```

npm i @types/bootstrap --save-dev

## extended-users-component.html

```
<app-user-info (eventPipe)="onEvent($event)"></app-user-info>
```

## extended- users-component.ts

```
onEvent(user:User) {
  this.users.push(user);
}
```

# Pridávanie/úprava používateľa

**app/users/users-service.ts**

```
import { HttpClient } from '@angular/common/http';  
...  
private restServerUrl: string = "http://localhost:8080/";  
  
public saveUser(user: User):Observable<User> {  
    return this.httpClient.post<User>  
        (this.restServerUrl + "users/" + this.token, user)  
        .pipe(catchError(error => this.processError(error)));  
}
```

# edit-user pre pridávanie aj editáciu

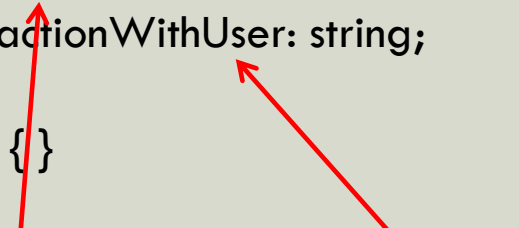
- Potrebujeme, aby rodičovský komponent informoval o tom, koho editujeme
- V `user-edit.component.ts` si zadefinujeme, ktoré inštančné premenné chce mať na vstupe
  - ▣ Rodičovský komponent nevolá konštruktor, to robí framework
  - ▣ premenná `user` (nový alebo editovaný),
  - ▣ premenná `actionWithUser` (text, čo sa deje)
- V `users.component.html` zadefinujeme hodnoty pre tieto premenné

# Odoslanie vstupu od rodiča

## user-edit-component.ts

```
import {Input, OnChanges} from '@angular/core';

export class UserEditComponent implements OnChanges {
  @Input() public user: User;
  @Input() public actionWithUser: string;
  ...
  ngOnChanges() {}
}
```

Two red arrows originate from the HTML template below. One arrow points from the `selectedUser` attribute value to the `@Input() public user` property in the TypeScript class. The other arrow points from the `actionWithUser` attribute value to the `@Input() public actionWithUser` property in the TypeScript class.

## extended-users-component.html

```
<app-user-info [user]="selectedUser" [actionWithUser]="actionWithUser"
(eventPipe)="onEvent($event)"></app-user-info>
```

- Ešte doplníme obslužné metódy pre udalosti, keď používateľ chce pridať alebo editovať používateľa, ktoré nastaví `actionWithUser` a `selectedUser`

# REST server - metódy

Ďalší preddefinovaní  
používatelia a ich heslá :

Lucia : lucia

John : john

Andrej : andre

- GET: /users
- POST: /login
  - ▣ V tele pošleme {"name": "Peter", "password": "upjs"}
  - ▣ príde vygenerovaný token
- GET: /users/{token}
  - ▣ prídú používatelia aj s neverejnými atribútmi
- GET: /user/{id}/{token}
- GET: /bg-user/{id}/{token}
  - ▣ vnútorná reprezentácia používateľ'a (obsahuje aj heslo)
- POST: /users/{token}
  - ▣ Cez POST pošleme JSON používateľ'a, ktorého chceme uložiť
- DELETE: /user/{id}/{token}

# REST server - metódy

- GET: /groups
- GET: /group/{id}
- POST: /groups/{token}
  - ▣ Cez POST pošleme JSON skupiny, ktorú chceme uložiť
- DELETE: /group/{id}/{token}

# Nastavenie @Input() parametrov

## extended-users-component.html (časť)

```
<tr *ngFor="let user of users" (dblclick)="editUserClick(user)">
```

## extended-users-component.ts

```
addUserButtonClick() {  
  this.selectedUser = new User("", "");  
  this.actionWithUser = "add";  
}
```

```
editUserClick(user: User) {  
  this.selectedUser = user;  
  this.actionWithUser = "edit";  
  $('#myModal').modal('toggle'); // zobrazí modálne okno  
}
```



# Vymazanie používateľa

## **app/users/users-service.ts**

```
import { HttpClient } from '@angular/common/http';  
...  
private restServerUrl: string = "http://localhost:8080/";  
  
public deleteUser(user: User):Observable<boolean> {  
    return this.httpClient.delete<boolean>  
        (this.restServerUrl + "user/" + user.id+ "/" + this.token)  
        .pipe(map(_ => true),  
            catchError(error => this.processError(error)));;  
}
```

# Angular – advanced (reklama)

- Angular Material
- Reaktívne formuláre a vlastné validátory
- Feature moduly, hierarchcké routovanie
- Strážcovia routovania
- NGXS (úložisko na ukladanie a monitoring stavových dát )
- Filtrovanie, paginácia, sort dát v klientovi aj na serveri
- WebSockets