



## 02. LABOR - ANGULAR

Segédlet a Szoftverfejlesztés laboratórium 2 c. tárgyhoz

Szabó Gábor, Kővári Bence

2019.

Frissítette: Erdős Szilvia, 2020.

### Szerzői jogok

Jelen dokumentum a BME Villamosmérnöki és Informatikai Kar hallgatói számára készített elektronikus jegyzet. A dokumentumot a Szoftverfejlesztés laboratórium 2 c. tantárgyat felvevő hallgatók jogosultak használni, és saját céljukra 1 példányban kinyomtatni. A dokumentum módosítása, bármely eljárással részben vagy egészben történő másolása tilos, illetve csak a szerző előzetes engedélyével történhet.



## BEVEZETÉS

### CÉLKITŰZÉS

A labor célja, hogy a hallgatók a webes fejlesztés kliens oldali részébe kissé mélyebb betekintést nyerjenek. A labor során az Angular keretrendszer használatával a hallgatók megvalósítanak egy egyszerű kliens-szerver kommunikációt, a korábbi alkalommal megismert ASP.NET Core alapok felhasználásával.

A laboron az ASP.NET Core laboron készítettéhez hasonló, ingatlanhirdetési felületet fogunk készíteni, ezennel kliens oldali renderelés (HTML összeállítás) segítségével.

### ELŐFELTÉTELEK

A labor elvégzéséhez szükséges eszközök:

- Microsoft Visual Studio 2017/19
  - .NET Core 2.1 SDK
- NgOtthonBazar\_kiindulo.zip
- Visual Studio Code
- NPM
  - Angular CLI eszköz (6.1.4) globálisan telepítve

### AMIT ÉRDEMES ÁTNÉZNE

- HTML, CSS, JS alapok
- .NET Core alapok
- Angular alapok, TypeScript

### A LABOR ÉRTÉKELÉSE

A labor teljesen önálló munka végzésére szolgál. A teljes munkát a laborvezető a labor végén értékeli. Az értékeléskor a laborvezető kérdéseket tehet fel az elkészült alkalmazással kapcsolatban, amik befolyásolhatják az érdemjegyet. Az értékelést a hallgató a labor végeztével, de legkésőbb a laborvezető által a labor megkezdésekor jelzett időpontban jelzi a laborvezetőnek. Az ellenőrzéshez segítséget nyújt a labor végén összeállított ellenőrző lista.

A labor végén a Moodle portálra feltöltendő az elkészült forráskód ZIP formátumban. A ZIP-ből törlendő a fordítási artefaktok és külső függőségek (bin, obj, packages, node\_modules mappák).

## SZERVEROLDAL ÁTTEKINTÉSE

A laborhoz tartozó *NgOtthonbazar\_kiindulo.zip* fájlban az alkalmazás szerveroldali komponense található. Ez az előző laboron készített ASP.NET Core alkalmazás átalakítva, hogy ne HTML-, hanem JSON alapú kommunikáció folyjék a kliens és a szerver között. A szervert egy REST API írja le.

A projekt felépítése az alábbi:

A Program.cs az alkalmazás belépési pontját tartalmazza, a webszerver konfigurációjának elvégzéséért és elindításáért felel.

A Startup.cs-ben található a DI IoC konténer konfigurációja (*ConfigureServices*) ill. az alkalmazás middleware-jeinek egymásra építése (*Configure*).

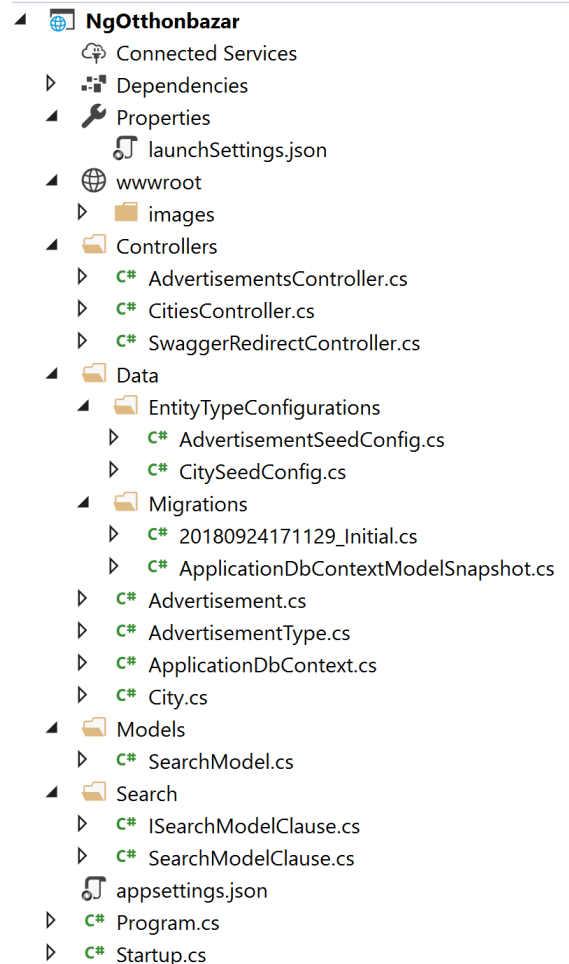
A Data mappában az adatmodellünk és az adatbázissal kapcsolatot teremtő kontextus osztályunk található. Itt találhatók a migrációk is, amelyekkel az adatbázissémát különböző verziók között módosíthatjuk és az ősfeltöltésért felelős entitás típusok konfigurációi.

A Models osztályban a kereséshez szükséges adatosztály található. A Search mappában a komplexebb keresési objektumleírókat lehet megtekinteni, amelyeket a szűréskor használunk fel.

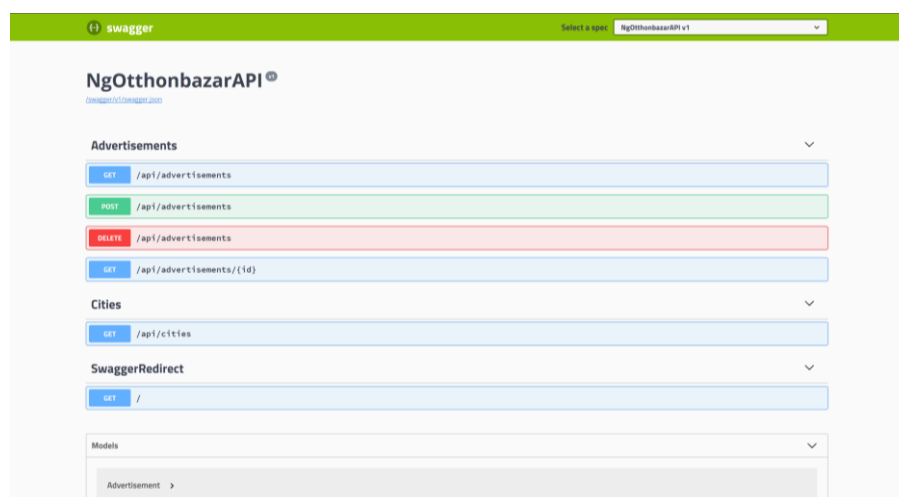
A wwwroot mappában a statikusan kiszolgálható fájlok találhatók, amik esetünkben a hirdetések képei.

Két controller szolgálja ki a kéréseket, az Advertisements- és a Cities controllerek, rendre a /api/advertisements és /api/cities végpontokon. A SwaggerRedirectController a gyöker URL-ről irányít át a /swagger URL-re.

Hozzuk létre a (localdb)\MSSQLLocalDB adatbázison az SQL Server Object Explorer segítségével az NgOtthonbazar nevű adatbázist (erre csak akkor van szükségünk, ha a laborgépeken a DB létrehozási jog meg van tagadva a futó alkalmazásnak)!



Indítsuk el a projektet **Ctrl+F5**-tel IIS Express-en!



A projekt publikálja a saját API végpontjait Swagger segítségével, így meg tudjuk tekinteni az alkalmazás API-ját a <http://localhost:10001/swagger> címen (ahová a megfelelő Controller átirányít minket indulás után).

Az URL-ek természetesen nem érzékenyek a kis-nagybetűkre.

## API TESZTELÉSE

A Swagger által generált oldalon tesztelhetjük az alkalmazás API-ját. Az egyes útvonalakra kattintva megnyílik a lehetőségünk AJAX kéréseket intézni a szerver felé.

Kérjük le a 3-as ID-jú hirdetés adatait a szervertől!

**Advertisement**Show/HideList OperationsExpand Operations

GET /api/Advertisement/Search

GET /api/Advertisement/Details

Response Class (Status 200)

Success

Model

Example Value

```
{
  "id": 0,
  "advertisementType": 0,
  "cityID": 0,
  "city": {
    "id": 0,
    "zip": "string",
    "name": "string"
  },
  "address": "string",
}
```

Response Content Type text/plain

Parameters

Parameter	Value	Description	Parameter Type	Data Type
id	<input type="text" value="3"/>		query	integer

Response Messages

HTTP Status Code	Reason	Response Model	Headers
404	Not Found		

Try it out!

Az API leírás segítségével láthatjuk a kliens-szerver kommunikációban használt modell típusokat is, amelyeket fel fogunk használni a kliensalkalmazásban.

Ne zárjuk be a Visual Studio-t, hogy a szerver a háttérben futhasson. Ha szükséges lesz, a szervert debugolni is lehetséges, ekkor F5 segítségével szükséges indítani az alkalmazást. Ha valaki a konzolos szerver (Kestrel) preferálja, a fenti menüben az IIS Express helyett az NgOththonbazar lehetőséget válassza, így a konzol lesz maga a szerveralkalmazás is, ami valós idejű naplózóként is működik.

## KLIENSALKALMAZÁS

A vékonykliens alkalmazásunk egy Angular keretrendszert használó Single Page Application lesz, amiben a hirdetések listázása, szűrése, létrehozása, szerkesztése és törlése lehetséges. A klienst nulláról készítjük el, hogy lássuk a fejlesztés teljes folyamatát.

Az önálló feladatok megoldásához sokat segítenek a hivatalos dokumentációs oldalak:

- Angular: <https://angular.io/>
- Bootstrap: <https://getbootstrap.com/docs/4.1/getting-started/introduction/>
- NgBootstrap: <https://ng-bootstrap.github.io/#/components>

Bár a gépünkre telepítve van az npm csomagkezelő, nem biztos, hogy elérési útvonalon van.

- Indítsunk egy parancssort (cmd.exe).
- Amennyiben a parancssor nem ismeri az „npm” parancsot, másoljuk be a következő utasítást:

```
setx PATH "%AppData%\npm;%PATH%"
```

- Hagyjuk figyelmen kívül az esetleges túl hosszú útvonalról szóló üzenetet és zárjuk be az ablakot. (Az elérési útvonal beállítását csinálhattuk volna elegánsabban is, de most nekünk ez elég)

Készítsük el a kiinduló alkalmazást:

- A szerverprojekt mappája mellett hozzunk létre egy NgOtthonbazarClient mappát!
- A mappán jobb klikkelve indítsuk el a VS Code szerkesztőt, vagy a szerkesztőt elindítva nyissuk meg a létrehozott mappát, mint munkakönyvtárat! (A laborgépeken lehet, csak az utóbbi működik)
- VS Code-ban a Ctrl+ö billentyűkombinációval, vagy a menüből nyissuk fel a Terminal ablakot. Ez egy Powershell konzolt indít a szerkesztőn belül.
- Telepítsük az Angular CLI-t (az utasítást innen puskázhatjuk ki: <https://cli.angular.io/>).

```
o npm install -g @angular/cli
```

- Adjuk ki az alábbi parancsot egy projekt létrehozásához (fontos a . karakter a parancs végén):

```
o ng new otthonbazar --directory .
```

- A feltett kérdésekre az alapértelmezett választ adjuk (Enter)
- Elképzelhető, hogy a laborgépeken az npm installálási fázisakor a generálás lefagy. Ekkor a hibát jelző ablak megjelenésekor kezdjük újra a folyamatot, de előtte töröljük a generálás által helyben létrehozott fájlokat. Ha többszörre sem sikerül, próbáljuk meg kiadni a parancsot CMD-ből ugyanebben a mappában.
- Ez legenerálja nekünk a kiinduló projektet, elképzelhető, hogy eltart néhány percig a futása. A forráskód az src mappában lesz, itt fogunk dolgozni.

- Amennyiben a telepítés hibára futna, vagy a hálózat miatt túl lassú (>5 perc) lenne, a tárgyhonlapon biztosítunk egy *NgOtthonbazar\_Packages.zip* fájlt is, mely már a műveletek végeredményét tartalmazza. Probléma esetén térjünk át ennek a használatára.

Az alkalmazás alapvető felépítése kapcsán az alábbiakat érdemes áttekintenünk:

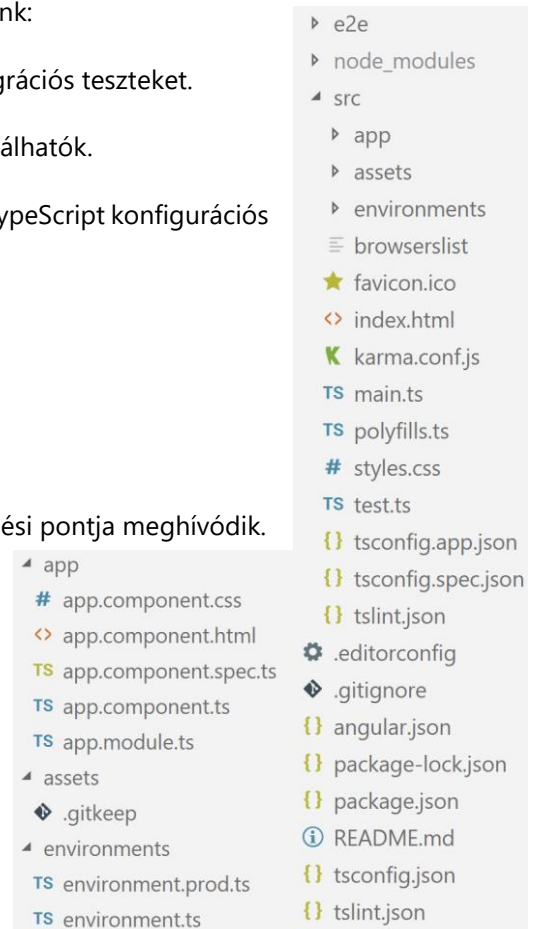
Az e2e mappa az end-to-end tesztelési projekt gyökere, itt készíthetünk integrációs teszteket.

A node\_modules mappában az alkalmazás NPM-ből származó függőségei találhatók.

Az angular.json fájl az Angular CLI, a package.json az NPM, a tsconfig.\*.json a TypeScript konfigurációs fájljai.

Az src mappa az alkalmazásunk forráskódja, ezen belül a fontosabb elemek:

- A styles.css fájlban a globális CSS szabályainkat definiálhatjuk.
- A main.ts az alkalmazás belépési pontja.
- Az index.html a klasszikus kezdőoldal, amivel az alkalmazásunk belépési pontja meghívódik.
- Az assets mappa a statikus állományaink (pl. logók, dokumentumok) helye.
- Az environments mappában környezetfüggő beállításokat (pl. prod/dev) definiálunk.
- Az app mappában a saját alkalmazásunk forráskódja van. Jelenleg ez egyetlen alkalmazásmodult (AppComponent) és egyetlen komponenst (AppComponent) tartalmaz.



Az alkalmazásban használni fogjuk a Http szolgáltatást a szerveroldali adatok lekérésére. Ehhez a CORS (Cross-Origin Resource Sharing) szabályok miatt reverse proxy-t fogunk használni, ami a kimenő kéréseinket át fogja irányítani egy másik hostra (mindkettő lokális). Hozzunk létre egy **proxy.conf.js** fájlt az app gyökerében, ahol definiáljuk a reverse proxy szabályunkat az alábbi tartalommal:





```
module.exports = [
  {
    "context": [
      "/api",
      "/images"
    ],
    "target": "http://localhost:10001",
    "secure": false
  }
]
```

Módsítsuk a package.json fájlban a scripts/start elem tartalmát, hogy megkapja a proxy konfigurációját is:

```
{
  "name": "otthonbazar",
  ...
  "scripts": {
    "ng": "ng",
    "start": "ng serve --proxy-config proxy.conf.js",
  }
}
```

```
...
},
...
}
```

Indítsuk el a kiszolgálást:

- `npm start`
  - Ez elindítja a fejlesztéshez használható szerver a **localhost:4200**-on (meghívja a package.json-ben definiált start szkriptünket, tehát az `ng serve` parancsot, aminek átadja a proxy beállításunkat). Ez a szerver nem azonos az ASP.NET Core-os szerverrel (ami elvileg jelenleg is a `localhost:10001`-en fut), ez nem futtat a klasszikus értelemben vett szerveroldali kódot, csak fejlesztésre szolgál, pl. ki tudja szolgálni a memóriában tárolt, TypeScriptből fordított JavaScript forrást.
  - A kiszolgáláson felül a folyamat folyamatosan fut, figyeli a forrásfájlokat, és módosulásuk esetén újrafordítja a szükséges modulokat és értesíti az éppen kapcsolódott böngészőpéldányokat, hogy töltsék újra magukat. Tehát ha a kódban mentünk, a kapcsolódott böngészők frissülnek. Előfordul, hogy a fordítás hibára fut, akár hibajelzés nélkül is, ekkor le kell állítanunk az aktuális folyamatot (pl. Ctrl+C a Terminal ablakban), majd újra `npm start`-tal indítani a kiszolgálást. A hibakeresés második lépcsője legyen minden esetben az, hogy az alkalmazást ilyen módon újraindítjuk.
  - Ha új parancsot kell kiadnunk, akkor indítsunk új terminál ablakot a Terminal jobb felső sarkában található + gombbal:  +   . Ha az aktuális terminált bezárjuk, a szerver leáll.
  - Fontos, hogy bármiféle hiba esetén, amikor az alkalmazás nem töltődik be vagy hibásan működik, nézzük meg a böngészőben a konzolon levő hibaüzenetet (F12)!

Indítsuk el a böngészőt, és navigáljunk a `localhost:4200`-ra!

Az alkalmazásunkban az alábbi komponensekre lesz szükségünk:

- *AppComponent*: ez a komponens az alkalmazás belépési komponense, a routing számára biztosítja a gyökér elemet.
- *ListComponent*: a hirdetések listázására szolgáló komponens.
- *FilterComponent*: a *ListComponent* nézetén megjelenő komponens, ami a különböző feltételek mentén képes szűrni az elérhető hirdetéseket.
- *DetailsComponent*: a részletek önálló nézete és logikája.
- *EditComponent*: a szerkesztés/létrehozás nézete és logikája.

## HTTP LEKÉRDEZÉSEK

Nagyobb alkalmazásokban érdemes tartani a felelősségi körök szétválasztását, és több rétegbe szervezni az appunkat. Esetünkben pl. érdemes lenne bevezetni egy saját szolgáltatásréteget, ami a HTTP kommunikációt végzi. Most idő hiányában nem vezetjük be ezt a réteget, hanem a lekérdezést közvetlenül a komponensünk kódjában végezzük el az Angular HttpClient szolgáltatásának segítségével.

Az app.module-ba importáljuk az Angular HttpClient modult. Az Angular modularizált: bár gyakori, de nem feltétlenül szükséges HTTP alapú kommunikációt végeznie az alkalmazásunknak (lehet például, hogy az alkalmazásunk csak WebSocketen kommunikál, és akkor nincs szükségünk a teljes Angular HTTP stackre sem).

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { HttpClientModule } from '@angular/common/http';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Definiáljuk a DTO interfészeinket, amelyek az adathordozásra szolgálnak majd. Ehhez segítséget nyújt a Swagger által generált példa JSON fájl. Hozzuk létre az **src/app/models.ts** fájlt az alábbi tartalommal:

```
export enum AdvertisementType {
  flat = 0,
  house = 1,
  holidayHouse = 2,
  buildingPlot = 3
}

export interface City {
  id: number,
  zip: string,
  name: string
}
```

Próbáljuk meg önállóan definiálni az **Advertisement** interfészt is a Swagger adatok felhasználásával! (A megoldást itt is megadjuk)

```
export interface Advertisement {
  id: number;
  address: string,
  advertisementType: AdvertisementType,
  buildDate: number,
  cityId: number;
  city: City;
  description: string,
  halfRoom: number,
  room: number,
  imageUrl: string
  price: number,
  size: number,
}
```



A típusok ismeretében már le tudjuk kérdezni a szervertől az elérhető hirdetéseket (le tudnánk dinamikus típussá is képezni őket, viszont ez ellenjavallt, mert nagyon nagy hibalehetőséget hordoz magában). Az AppComponent-ben, annak inicializálásakor kérjük le a szervertől a hirdetéseket, és jelenítsük meg azokat:

```
import { Component, OnInit } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Advertisement } from './models';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  constructor(private http: HttpClient) { }
  advertisements: Advertisement[];
  ngOnInit() {
    this.http.get<Advertisement[]>('/api/advertisements').subscribe(resp =>
      this.advertisements = resp
    );
  }
}
```

A HttpClient szolgáltatás *Observable*-ökkel tér vissza, amelyek eseményfolyamnak tekinthetők. A `get()` kérésünk egy egyszer teljesülő (vagy hibára futó) eseményfolyam. Siker esetén a `subscribe()` függvényben megadott callback fut le, ami beállítja a komponens `advertisements` tulajdonságát a válaszban érkezett JSON-re. Nem történik meg sehol az `Advertisement` típus példányosítása, ugyanis a JSON csak „sima” objektumokat hoz létre az adott tulajdonságokkal. Tehát nem érdemes megpróbálni logikát elhelyezni ezekben az elemekben, ugyanis valójában sosem lesznek az adott típusok ténylegesen példányosítva. A JS sajátosságaiból fakad a dinamikus típusossága, amit a TypeScript fejlesztési időben el tud fedni, de futási időben nem.

Cseréljük le az AppComponent nézetét az `app.component.html`-ben az alábbira:

```
<div *ngFor="let advertisement of advertisements" style="border: 1px solid blue">
  <p>{{advertisement.city.name}} ({{advertisement.city.zip}}, {{advertisement.cityId}})</p>
  <p>{{advertisement.address}}</p>
  <p>{{advertisement.advertisementType}}</p>
  <p>{{advertisement.buildDate}}</p>
  <p>{{advertisement.description}}</p>
  <p>{{advertisement.halfRoom}}</p>
  <p>{{advertisement.room}}</p>
  <p>{{advertisement.imageUrl}}</p>
  <p>{{advertisement.price}}</p>
  <p>{{advertisement.size}}</p>
</div>
```

*Tipp: inline stílust (a HTML elemen elhelyezett style attribútummal) csak kivételes esetekben alkalmazzunk! A példában az egyszerűség kedvéért adtuk meg a stílust a HTML-ben, de mindig szervezzük a CSS-ünket saját strukturált CSS (LESS/SCSS) fájlokba, ill. használjuk a keretrendszerünk adottságait, pl. a komponens alapú stílusozást!*

Ez a kifejezés bejárja az `advertisements` tulajdonságát a komponensnek (amennyiben az nem `undefined`, tehát amikor megérkezik az adat JSON-ként a szervertől és beállítjuk), és az `advertisement` iterátor változóban tárolja el azt. A `<div>`

elemen és azon belül ezt a változót tudjuk használni. Az adatkötésbe magának a hirdetésnek az egyes tulajdonságait tesszük, így a felületen meg fog jelenni az adat, ahogyan a szerverről érkezett.

## ALKALMAZÁS STÍLUSOZÁSA

Ahhoz, hogy kicsit jobban tudjuk formázni az alkalmazást, szükségünk lesz a Bootstrap stílusozó keretrendszerre. Az alkalmazás kinézetének az előző laboron készített, Razorben renderelt alkalmazást vesszük alapul.

Sok JavaScript alapú keretrendszer képes integrálódni az Angularrel, a Bootstrap is ilyen, viszont jellemzően az adatkötések megvalósításához burkoló komponenseket írnak. Ehhez az NgxBootstrap-et fogjuk használni, amit telepítenünk kell. Egy új Terminalban adjuk ki az alábbi parancsokat:

```
ng add @ng-bootstrap/ng-bootstrap@7.0.0  
  
ng add bootstrap@4.5.2
```

Ez telepíti a node\_modules mappába az ng-bootstrap modult, valamint magát a Bootstrap-et, amit az alkalmazásunkba kell kötnünk. Ennek segítségével egyszerűen tudjuk kezelni a Bootstrap komponenseit kódból és template-ből is (modális ablakok, navigációs sáv stb.).

A stílusozáshoz a saját alkalmazásmodulunkhoz (AppModule) hozzá kell rendelnünk az NgbBootstrapModule-t a HttpClientModule-hoz hasonlóan, ami a számunkra szükséges direktívákat, komponenseket, szolgáltatásokat tartalmazza:

```
import { BrowserModule } from '@angular/platform-browser';  
...  
import { NgbModule } from '@ng-bootstrap/ng-bootstrap';  
  
@NgModule({  
  ...  
  imports: [  
    BrowserModule,  
    HttpClientModule,  
    NgbModule  
  ],  
  ...  
})  
export class AppModule { }
```

Ez még csak a JavaScript funkcionalitást tartalmazza a Bootstrapből, a stílusok használatához szükséges a Bootstrap CSS-ének betöltése is. Angular CLI esetén ezt az **angular.json** fájlban adjuk meg, a JSON-ben az **app/styles** útvonalon található elem tartalmát egészítsük ki a Bootstrap CSS útvonalával legyen az alábbi:

```
"styles": [  
  "src/styles.css",  
  "node_modules/bootstrap/dist/css/bootstrap.css"  
],
```

Ezután szükséges újraindítanunk az Angular CLI szerveret, ugyanis a konfiguráció változását nem érzékeli. A Terminalban, ahol a szerverünk fut, **Ctrl+C, Y**, majd **Enter** hatására a szerver leáll. Indítsuk újra:

```
npm start
```

Másoljuk be az **app.component.html** új tartalmát, ami az oldal vázát adja majd:

```
<div class="container">
  <nav class="navbar navbar-expand-lg navbar-light bg-light">
    <a class="navbar-brand">Otthonbazar</a>
    <button class="navbar-toggler" type="button">
      <span class="navbar-toggler-icon"></span>
    </button>

    <div class="collapse navbar-collapse">
      <ul class="navbar-nav mr-auto">
        <li class="nav-item active">
          <a class="nav-link">Hirdetések</a>
        </li>
        <li class="nav-item">
          <a class="nav-link">Hirdetés feladása</a>
        </li>
      </ul>
    </div>
  </nav>
  <!-- TODO: Ide jön a tartalom -->
  <hr />
  <footer>
    &copy; 2020 - Otthonbazar
  </footer>
</div>
```

A fenti HTML tartalom egyelőre teljesen statikus. Kis képernyőkön a „hamburger” menüre kattintva a jobb felső sarokban nem történik semmi, pedig meg kellene jelennie a navigációs elemeknek. Az ASP.NET Core alkalmazásunkban a beépített Bootstrap „unobtrusive” módon működött, azaz JavaScript kód írása nélkül képes volt kezelni ezeket azáltal, hogy deklarátívan összekötöttük a HTML elemeket a megfelelő attribútumokkal.

Most a template-ben imperatív megközelítést fogunk alkalmazni. A gomb klikk eseménykezelőjére iratkoztassunk fel egy kezelőt, ami a navbarCollapsed változóját fogja negálni minden kattintással a komponensnek, a header alatt levő div pedig ettől függően lesz látható vagy rejtett:

```
<nav ...>
  <a class="navbar-brand">Otthonbazar</a>
  <button class="navbar-toggler" type="button" (click)="navBarCollapsed = !navBarCollapsed">
    ...
  </button>
</div>
<div class="collapse navbar-collapse" [ngbCollapse]="navBarCollapsed">...</div>
```

Az eseménykezelőt ( $V \rightarrow M$  irányú adatkötés) az `(esemény) = "kezelő"` szintaxissal regisztráljuk az elemre, a modell változására reagáló ( $M \rightarrow V$ ) adatkötést pedig a `[tulajdonság] = "modellérték"` szintaxissal adjuk meg. Mindkét adatkötési irány lehetséges a DOM elemre értelmezhető eseményre/tulajdonságra, vagy az elemen megadott valamely *direktíva* eseményre/tulajdonságra. Itt az utóbbit használjuk, a collapse nem a HTMLDivElement tulajdonsága, hanem a Bootstrap CollapseDirective tulajdonsága, amelyet az elemen akkor helyeztünk el, amikor a `[collapse]` attribútumot elhelyeztük.

Egy hibát viszont vétettünk: a navigációs sáv alapértelmezetten nyitva van. Miért látható alapértelmezetten nyitott állapotban a navigációs sáv? A `navBarCollapsed` változó kezdetben nem létezik a komponensen. Amikor kattintunk, akkor a változónak új értéket adunk, az eredeti érték negáltját: `!undefined === true`, tehát ekkor már az érték igaz lesz. Az `ngbCollapse` kötést viszont ennek negáltjára vizsgáljuk.

Gyors megoldásnak tűnne negálni a kötést, viszont akkor a változó neve nem korrelálna az implikált működéssel. A helyes megoldás a változót a komponens kódjában deklarálni (amit egyébként is illik megtennünk), és `true` kezdőértéket adni neki:

```
export class AppComponent {  
  navBarCollapsed = true;  
}
```

## ROUTING

Most, hogy készen vagyunk az alkalmazás vázával, adjuk hozzá a későbbiekben szükséges komponenseket:

```
ng generate component list
ng g c details
ng g c edit
ng g c filter
```

Láthatjuk, hogy az alkalmazás app mappáján belül a komponensek saját mappákba kerülnek, és a komponensek regisztrációja az AppModule dekorátorának declarations tömbjébe kerül.

Ahhoz, hogy a komponenseket aktiválni tudjuk, útvonalakat kell hozzájuk rendelnünk. Az útvonalak az URL alapján hierarchiába szervezettek, így a komponensek is egy adott hierarchiát reprezentálnak.

Nem minden komponenst fogunk saját útvonalhoz rendelni, a FilterComponent például kizárólag a ListComponent template-jében fog szerepelni, kézzel helyezzük el a template-ben.

A routing konfigurációját az **@angular/router**-ben található **RouterModule** segítségével tudjuk elvégezni. Az AppModule kódjában adjuk meg a routing szabályokat, és regisztráljuk az elérhető útvonalakat:

```
...
import { Route, RouterModule } from '@angular/router';
const routes: Route[] = [
  { path: '', component: ListComponent },
  { path: 'details/:id', component: DetailsComponent },
  { path: 'edit/:id', component: EditComponent },
  { path: 'create', component: EditComponent }
];

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

Tehát a gyöker URL-en az AppComponent példányosodik (ezt az @NgModule dekorátorban megadott bootstrap property-ből tudjuk) az index.html-ben megadott helyen, aminek gyerekei az egyes útvonalakon érhetők el. Azok a komponensek, amelyeknek gyerek útvonalai / komponensei is vannak, a <router-outlet> komponens segítségével helyezhetők el a szülő komponens kódjában. A Details és Edit komponensek várnak még egy id paramétert is az URL-ből.

Tegyük tehát az AppComponent template-jébe a RouterOutlet komponenst, ahol szeretnénk a navigált elemeket látni:

```
<div class="container">
  <nav ...></nav>
  <router-outlet></router-outlet>
</div>
<hr />
<footer>
  <p>&copy; 2018 - Otthonbazar</p>
```

```
</footer>
</div>
```

Ennek eredményeképpen a gyökér URL-en a ListComponent található, viszont nem tudunk a többi komponensre navigálni, csak ha az URL-t megfelelőképpen változtatjuk meg.

Adjuk meg a fenti menüsávon az egyes elemekhez, hogy melyik útvonalra navigáljanak! Ne használjuk a nyilvánvalónak tűnő href attribútumot, hanem használjuk helyette a Router modulban található RouterLink direktívát! A href attribútum a megszokott módon, teljes oldalújrátöltéssel töltene be az oldalt. Az app.component.html-ben adjuk meg a navigációs elemekben, hogy hova navigáljanak:

```
<nav class="navbar navbar-expand-lg navbar-light bg-light">
  <a routerLink="{{['/']}}" class="navbar-brand">Otthonbazar</a>
  <button class="navbar-toggler" type="button" (click)="navBarCollapsed = !navBarCollapsed">
    <span class="navbar-toggler-icon"></span>
  </button>

  <div class="collapse navbar-collapse" [ngbCollapse]="navBarCollapsed">
    <ul class="navbar-nav mr-auto">
      <li class="nav-item active">
        <a [routerLink]="['/']" class="nav-link">Hirdetések</a>
      </li>
      <li class="nav-item">
        <a routerLink="create" class="nav-link">Hirdetés feladása</a>
      </li>
    </ul>
  </div>
</nav>
```

A három szintaxis különböző, mégis nagyon hasonló: a `[routerLink]="[routePart1, routePart2, ...]"` szintaxis a jobb oldali paraméterekből összefűzött útvonalra navigál. Akkor hasznos ezt a kötést használnunk, ha a jobb oldali kötés nem konstans értékhez köt, itt használhatunk adatkötést interpoláció nélkül. Esetünkben ez a második eset valójában ekvivalens az első kötéssel, ami interpolációt használ: `routerLink="{{[routePart1, routePart2, ...] }}"`.

A harmadik esetben a tulajdonságot a jobb oldalon megadott konstans string értékhez kötjük. **Fontos**, hogy ez a két szintaxis nem a routerLink sajátja, hanem bárhol használhatjuk ezt a szintaxist adatkötéshez. Vegyük észre a hasonlóságot a tényleges attribútumokkal: amikor attribútumnak értéket adunk, konstans értéket kap. Ez tehát az egyirányú adatkötés egy speciális formája, amikor nem frissül az adatkötött érték. Ami a routerLink számára specifikus viszont, hogy átadhatunk neki tömböt az URL darabokkal, vagy egy stringet is a teljes vagy relatív URL-lel.

Az aktuális útvonalat jellemzően CSS osztállyal szokták ellátni, erre használhatjuk a RouterLinkActive direktívát. Ha adott DOM elemre tesszük, amin (vagy gyerekelemén) routerLink is található, akkor az osztály akkor kerül az adott DOM elemre, ha az routerLink által adott útvonalon vagyunk. Ha más elemen szeretnénk használni, akkor a direktívát exportáljuk egy template változóba, és azt használjuk fel máshol a template-ben.

```
<li class="nav-item" routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">
  <a [routerLink]="['/']" class="nav-link">Hirdetések</a>
</li>
<li class="nav-item" [class.active]="createActive.isActive">
```

```
<a routerLink="create" routerLinkActive #createActive="routerLinkActive"
  class="nav-link">Hirdetés feladása</a>
</li>
```

A fenti szintaxis a # szimbólummal a template változó létrehozása, amelynek két módja van: ha nem kap paramétert, akkor a változóban egyszerűen eltároljuk az adott DOM elemet. Ha kap paramétert, akkor a DOM elemen elérhető, adott néven exportált direktívát kapja értékül (ezért tettük fel a direktívát végrehajtó attribútumot, hogy exportálhassuk a szükséges példányt). A RouterLinkActiveOptions direktíva kiegészíti a RouterLinkActive-ot, az első link nekünk pontosan akkor kell, hogy aktív legyen, ha pontos az URL egyezés, ellenkező esetben prefix alapú, ezért mindig aktív volna.

A navigációval készen vagyunk, készítsük el a listázás felületet!

## LISTÁZÁS

A listázáshoz szükségünk van a szerverről érkező adatokra. Eddig az AppComponent kérte le a szervertől az adatokat, viszont ez a funkcionalitás a listázó komponens feladata. *Helyezzük át* a lekérdezést megvalósító kódot a listázó komponensbe, és tegyük saját függvénybe:

```
export class ListComponent implements OnInit {
  constructor(private http: HttpClient) { }
  advertisements: Advertisement[];
  ngOnInit() {
    this.getAdvertisements();
  }
  getAdvertisements() {
    this.http.get<Advertisement[]>('/api/advertisements')
      .subscribe(resp => this.advertisements = resp);
  }
}
```

A komponens advertisements tulajdonságában tehát megtalálhatók lesznek a hirdetések. Járjuk be a hirdetéseket, és helyezzük el őket a DOM-ban (a listázó komponens template-jében):

```
<div class="row">
  <div class="col-md-4">
    <app-filter></app-filter>
  </div>

  <div class="col-md-8">
    <h2>Hirdetések</h2>

    <div class="row">
      <div class="col-lg-6" *ngFor="let advertisement of advertisements">
        <a [routerLink]="['details', advertisement.id]">
          <img class="img-fluid rounded mx-auto d-block"
            [src]="advertisement.imageUrl || 'https://via.placeholder.com/340x255'" />
        </a>

        <div class="row">
          <h4 class="col-6">{{advertisement.price}} MFt</h4>
          <h4 class="col-6 text-right">{{advertisement.size}}&nbsp;m&nbsp;m&nbsp;m<sup>2</sup></h4>
        </div>
        <p class="pull-left">
          {{advertisement.city?.zip}} {{advertisement.city?.name}}
        </p>
      </div>
    </div>
  </div>
```

```

<p class="pull-right">{{advertisement.address}}</p>

<p>{{advertisement.advertisementType}}</p>

<dl>
  <dt class="pull-left">Szobák</dt>
  <dd class="pull-left">{{advertisement.room}} + {{advertisement.halfRoom}}</dd>
</dl>
</div>
</div>
</div>
</div>

```

A lakás típusát nem tudjuk szépen megjeleníteni. Érdemes készítenünk egy pipe-ot erre a feladatra. Pipe-ot jellemzően nyers adatok felületen történő megjelenítésére, formázására használunk, de léteznek egyéb pipe-ok is (pl. az AsyncPipe, ami a paraméterül kapott Observable példányt bevárja, az eredményt kiburkolja).

```
ng generate pipe advertisement-type
```

Generálódik egy osztály, ami be is lesz regisztrálva az alkalmazásmodulunkba. Az így generált osztály transform függvényét kell megvalósítanunk:

```

export class AdvertisementTypePipe implements PipeTransform {
  private static lookup = ["Lakás", "Ház", "Nyaraló", "Építési telek"];

  transform(value: AdvertisementType): string {
    return AdvertisementTypePipe.lookup[value];
  }
}

```

A pipe-ot ezután a template-ben használhatjuk:

```
<p>{{advertisement.advertisementType | advertisementType}}</p>
```

## RÉSZLETEK

A részletekre a hirdetés képére kattintva juthatunk.

A komponens példányosodása után le kell kérnünk a szerverről az adott ID-hoz tartozó hirdetést, hogy meg tudjuk jeleníteni annak adatait. Az ActivatedRoute szolgáltatás az aktuális komponenshez tartozó útvonalakról publikál információkat, fel tudunk iratkozni így a paraméterekre is, amiből az azonosítót elkérjük, majd a szervertől lekérdezzük a hirdetést.

```

import { Component, OnInit } from '@angular/core';
import { ActivatedRoute } from '@angular/router';
import { Advertisement } from '../models';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-details',
  templateUrl: './details.component.html',
  styleUrls: ['./details.component.css']
})
export class DetailsComponent implements OnInit {
  advertisement: Advertisement;
  constructor(private route: ActivatedRoute, private http: HttpClient) { }
}

```



```

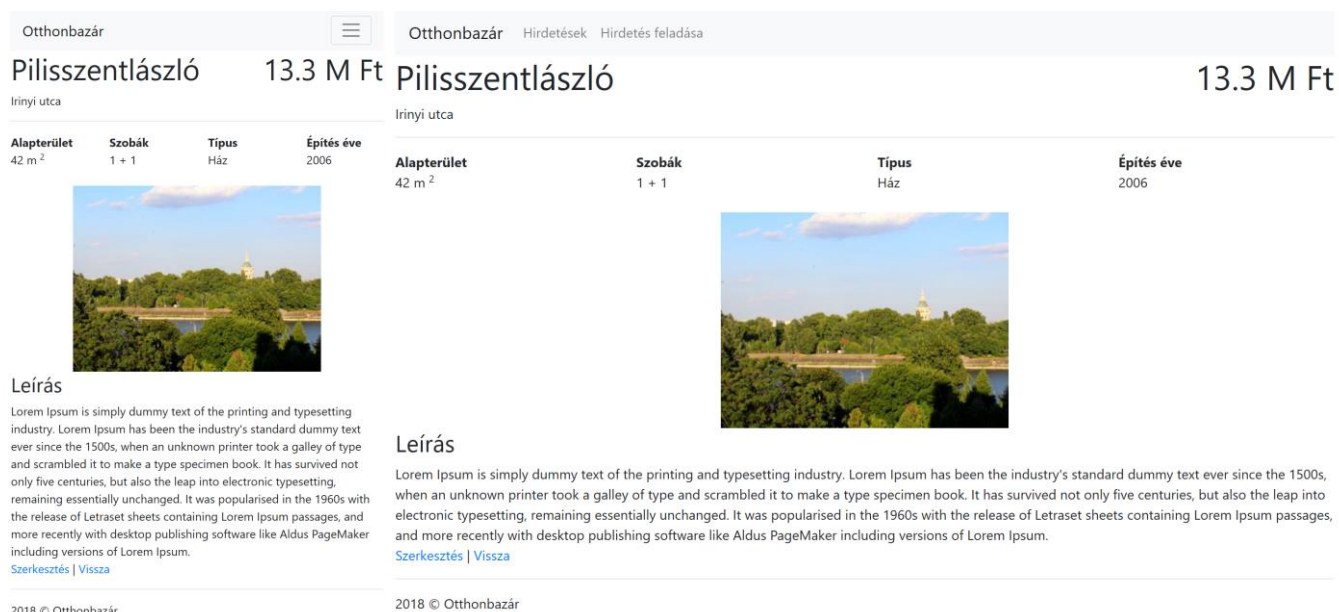
ngOnInit() {
  this.route.params.subscribe(params => {
    this.http.get<Advertisement>('api/advertisements/' + params['id'])
      .subscribe(resp => this.advertisement = resp);
  });
}
}

```

A korábbi feladatokhoz hasonlóan készítsd el a hirdetés részletes oldalát!

- Amíg nem érkezett meg a szerverről a hirdetés, addig ne kössük ki a tulajdonságait, mert hibát fogunk kapni! Használjuk az `<ng-container>` komponenst, hogy logikai egységbe fogjuk a teljes komponensünket, és egy helyen tudjuk meggátolni a hibás renderelést egy jól elhelyezett `*ngIf` direktívával!
- A layouthoz használjuk a Bootstrap dokumentáció segítségét. A legfontosabb, amire ügyeljünk, hogy row elemekben csak col elemek legyenek, és col elemek csak row elemekben legyenek!
- A szerkesztésre mutató linknél fontos, hogy relatív vagy abszolút URL-t használunk a navigációhoz!

A részletes oldal hasonlítson az alábbi képernyőképekre (vagy legyen egyéb módon, de igényes elrendezésű Bootstrap osztályok segítségével):



## SZŰRÉS

Valósítsuk meg a szűrést!

A szűrést a `FilterComponent` fogja kezelni, viszont a szűrőfeltételeket a `ListComponent`-nek is ismernie kell, ugyanis a lekérdezést továbbra is ő fogja végezni. Definiáljuk a szűrési feltételeket reprezentáló modellt, és a `ListComponent` által létrehozott üres példányt adjuk át a `FilterComponent`-nek, hogy tudjon rajta dolgozni! A `models.ts`-ben vegyük fel a `Filter` interfészt:

```

export class Filter {
  cityId?: number;
  cityName?: string;
}

```

```

priceMin?: number;
priceMax?: number;

sizeMin?: number;
sizeMax?: number;

roomMin?: number;
roomMax?: number;

page?: number;
pageSize?: number;
}

```

A ListComponent kódjában adjuk meg a szűrőt, illetve a szűrőt adjuk át a HTTP kérésnek paraméterül:

```

import { Filter } from '../models';
...
export class ListComponent implements OnInit {
  filter: Filter = new Filter();
  getAdvertisements() {
    this.http.get<Advertisement[]>('/api/advertisements',
      { params: new HttpParams({ fromObject: <any>this.filter }) })
      .subscribe(resp => this.advertisements = resp);
  }
  ...
}

```

Adjuk át a példányt a FilterComponentnek, ehhez neki bemeneti paraméterként várnia kell a Filtert, ill. publikálni fogja azt az eseményt, hogy a szűrés Keresés gombjára kattintottak:

```

import { Filter } from '../models';

export class FilterComponent implements OnInit {
  @Input() filter: Filter;
  @Output() submit: EventEmitter<void> = new EventEmitter<void>();
  ...
}

```

A konkrét átadás a ListComponent template-jében történik, valamint itt kötjük be az eseménykezelőt a submit esemény bekövetkeztére is:

```

<div class="col-sm-4">
  <app-filter [filter]="filter" (submit)="getAdvertisements()"></app-filter>
</div>

```

Ahhoz, hogy a szűrőben modell alapon tudjunk dolgozni, az alkalmazás moduljához hozzá kell rendelnünk a FormsModule-t, ami többek között az NgModel és NgForm direktívákat tartalmazza.

```

...
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [
    ...
    FormsModule
  ],
}

```

```

    providers: [],
    bootstrap: [AppComponent]
  })
  export class AppModule { }

```

Ezután már használhatjuk az NgModel-t, így elkészíthetjük a szűrőt magát. A FilterComponent template-jébe írjuk az alábbi kódot:

```

<h2>Keresés</h2>
<hr />

<form (ngSubmit)="submit.emit()">
  <div class="row my-2">
    <div class="col">
      <input [(ngModel)]="filter.cityName" name="cityName" type="text" class="form-control"
autocomplete="off" placeholder="Város" />
    </div>
  </div>
  <div class="row my-2">
    <div class="col-5">
      <input [(ngModel)]="filter.priceMin" name="priceMin" type="number" class="form-control"
autocomplete="off" placeholder="Ár (min Mft)" />
    </div>
    <div class="col-2 text-center">
      &ndash;
    </div>
    <div class="col-5">
      <input [(ngModel)]="filter.priceMax" name="priceMax" type="number" class="form-control"
autocomplete="off" placeholder="Ár (max Mft)" />
    </div>
  </div>
  <div class="row my-2">
    <div class="col-5">
      <input [(ngModel)]="filter.sizeMin" name="sizeMin" type="number" class="form-control"
autocomplete="off" placeholder="Méret (min m2)" />
    </div>
    <div class="col-2 text-center">
      &ndash;
    </div>
    <div class="col-5">
      <input [(ngModel)]="filter.sizeMax" name="sizeMax" type="number" class="form-control"
autocomplete="off" placeholder="Méret (max m2)" />
    </div>
  </div>
  <div class="row my-2">
    <div class="col-5">
      <input [(ngModel)]="filter.roomMin" name="roomMin" type="number" class="form-control"
autocomplete="off" placeholder="Szobák száma (min)" />
    </div>
    <div class="col-2 text-center">
      &ndash;
    </div>
    <div class="col-5">
      <input [(ngModel)]="filter.roomMax" name="roomMax" type="number" class="form-control"
autocomplete="off" placeholder="Szobák száma (max)" />
    </div>
  </div>

  <div class="row">
    <div class="col">
      <button type="submit" class="btn btn-default w-100">Keresés</button>
    </div>
  </div>
</form>

```

A szűrő komponens tehát a bementként várt Filter példányt manipulálja, és az űrlap elküldésekor elsüt egy eseményt, amire a ListComponent fel van iratkozva, és a megadott szűrővel új keresést indít a szerver felé.

## OPCIONÁLIS FELADAT: LÉTREHOZÁS ÉS SZERKESZTÉS, LAPOZÁS ÉS URL PARAMÉTEREK

### 2 IMSc pontért az alábbi feladatok közül legalább az egyiket el kell készíteni:

- Lapozás és URL paraméterek
  - o A Bootstrap és NgBootstrap lehetőségeit használva készítsd el a lapozó megoldást, ami az oldal alján és tetején is megtalálható! Ehhez a szerveren egy burkoló objektumot kell készítened, amihez az IPagedList GetMetaData() metódusa által visszaadott értéket is feldolgozod.
  - o Minden szűrés esetén az aktuális oldal URL-je tartalmazza a query stringben az API-hoz menő query string paramétert!
- Létrehozás és szerkesztés
  - o Készítsd el a szerkesztő felületet, ahol új hirdetés létrehozására és meglevő hirdetés szerkesztésére is van lehetőség! A felületen megjelenendő elemek azok, amelyeket a szerver vár, így a Swagger UI-on (<http://localhost:51056/swagger/>) meg tudod nézni a szükséges adatokat, ill. szükség esetén a szerver kódját is megvizsgálhatod. Kép feltöltése nem szükséges.
  - o Az irányítószám megadásakor automatikusan ki kell töltened a város nevét is a szerver oldalon található API végpont segítségével. A város neve legyen Typeahead vagy Autocomplete mező, ami szintén a szervertől várja a lehetséges városok listáját.
  - o A felületen egy űrlapon lehessen megadni az adatokat. A létrehozást a HttpClient service-en elérhető Post művelettel tudod elvégezni. A szerkesztés a létrehozástól annyiban különbözik, hogy a POST üzenetben szükséges megadni a szerkesztendő hirdetés azonosítóját is.

## A LABOR ÉRTÉKELÉSE

A labor során végzett munka értékeléseként az alábbiakat szükséges megvizsgálni:

### Elégséges szint:

Szerver oldal:

- Létrejött-e az NgOtthonbazar adatbázis az űs adatokkal, fut a szerver
- Elérhetők az API endpointok a Swagger UI-on keresztül

Alaparchitektúra:

- Elkészült az Advertisement modell interfésze
- Működik a HTTP kérések proxy-zása az alkalmazásszerver felé
- Láthatók a Bootstrap stílosozási megoldása, pl. a Navbar, tehát a Bootstrap CSS importálva van
- A menüsáv mobil nézeten alapértelmezetten zárva, de kattintásra kinyitható a menü

### Közepes:

Útvonalválasztás:

- Létrejött a 4 egyedi komponens (List, Details, Edit és Filter)
- Működik az alkalmazásban a navigáció az összes komponensre
- Az aktív útvonalnak megfelelő menüpont van kiválasztva a menüben, a listázó menüpont csak a főoldalon aktív

### Jó:

Listázás:

- Működik a hirdetések lekérdezése, megjelenítése
- Az üres szűrő komponens megjelenik
- A hirdetés típusa szövegesen, magyarul jelenik meg

Részletes oldal:

- A részletes oldalon megjelenik a hirdetés minden adata, igényes elrendezésű

### Jeles:

Szűrés:

- A szerveroldali szűrés működik, a gomb lenyomására a felületen frissülnek a feltételeknek megfelelő elemek

**IMSc pontért:**

Lapozás:

- A szűrőtől külön, a listázó alján és tetején is egy funkcionális lapozó található, ami a Bootstrap komponenseinek felhasználásával készült, csak annyi oldalt jelenít meg, amennyi létezik is

Létrehozás és szerkesztés:

- Létre lehet hozni új, és lehet szerkeszteni már meglevő elemet egy igényes, érthető felületen
- Az irányítószám kitöltésekor a város kitöltődik, a város nevének begépelésekor Autocomplete jellegű működés segíti a város kiválasztását