

Angular 16+ programozás

v1



Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Interaktív UI létrehozása

4. Adatkötés a komponensekben

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine

Bevezetés



Tanfolyam során használt infrastruktúra, szükséges eszközök:

1. Node.JS (npm csomagkezelő)

Node \geq v20.11.1, npm \geq 10.2.4

2. Visual Studio Code

3. Firefox (legalább v72) és Google Chrome (legalább v79) böngészők



- Az Angular egy nyílt forráskódú, frontend keretrendszer, amelyet a Google és a közösség fejleszt és karbantart.
- A keretrendszer célja, hogy segítse a fejlesztőket dinamikus, egyoldalas webalkalmazások (SPA) készítésében.
- Eredetileg 2010-ben jelent meg AngularJS néven, és 2016-ban újraírták Angular 2 néven, a TypeScript alapú architektúra felhasználásával, ami ma már egyszerűen Angularként ismert.

Angular jellemzők



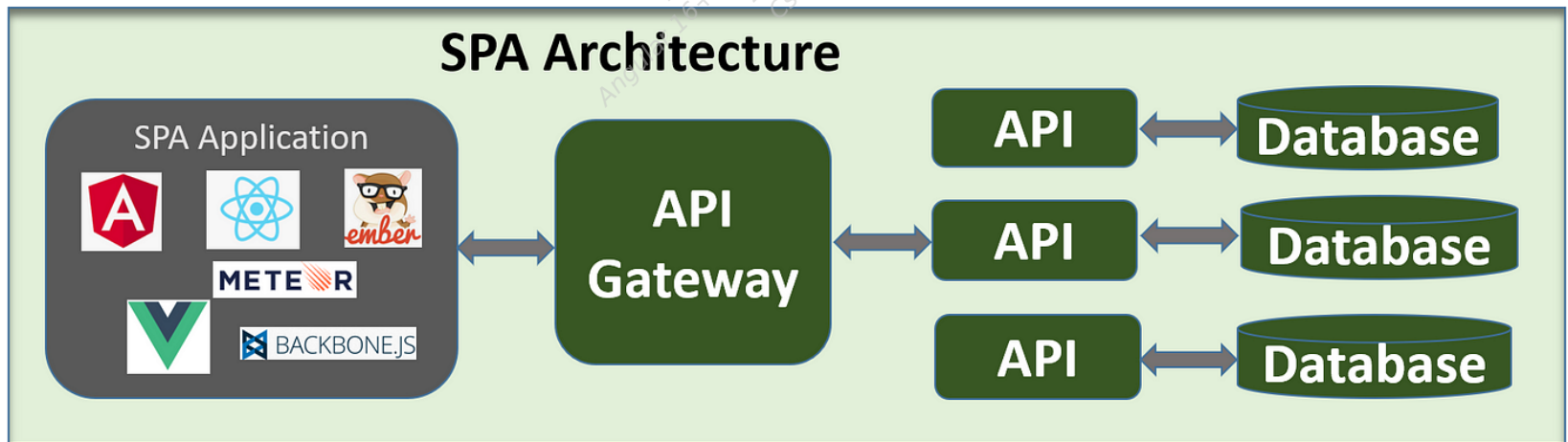
- Komponens-alapú architektúra.
- TypeScript programozási nyelv használata
- Egyoldalas alkalmazások (SPA) létrehozása
- Teljesítmény és skálázhatóság:
 - lazy loading,
 - server-side rendering
 - progressive web apps (PWA)
- Erős közösségi támogatás.
- Támogatja a teljes fejlesztést a létrehozástól a tesztelésig.
- Számos kiegészítő könyvtár.
- Mobil- és asztali fejlesztésre is használható.



Single Page Application (SPA)



- Egyetlen weboldal betöltése és a nézetek és adattartalom dinamikus cseréje az oldal egyes részein.
 - gyorsabb teljesítmény, jobb felhasználói élmény
 - könnyebb karbantarthatóság



TypeScript



- A TypeScript egy nyílt forráskódú nyelv, amelyet a Microsoft fejlesztett.
- Bármely érvényes JavaScript kód érvényes TypeScript kódként is működik, de további szintaktikai és típusellenőrzési képességeket ad hozzá.
- Előnyei:
 - Statikus típusrendszerrel rendelkezik, így a típusellenőrzést megvalósítja.
 - Megvalósítja az objektumorientált programozás paradigmát és alapelveket.

TypeScript típusok



- Primitív típusok: egy időben egy érték tárolására alkalmasak
 - number: szám típus (egész és tört)
 - string: szöveg típus
 - boolean: logikai típus
- Speciális típusok:
 - any: típusellenőrzés tiltása (dinamikus típus)
 - void: visszatérés nélküli alprogramok
 - unknown: ismeretlen típus
 - null: definiálatlan típus (csak null értéke lehet)
 - never: jelzi, hogy nem lehet visszatérési érték

TypeScript összetett adattípusok



- **array**: elemek sorszámozott tömbje, a tömb elemei azonos típusúak
- **tuple**: olyan tömb, amelye mérete kötött, és az egyes indexen lévő értékek eltérő típusúak
- **enum**: felsorolás típus, ha az értékek listája felsorolással megadható
- **objektum**: kulcs-érték pár, ahol az érték lehet adattartalom vagy metódus
- **unió típus**: ha egy érték több eltérő típust is felvehet, akkor a lehetséges típusok listáját | (pipe) jellel soroljuk fel

Típusmegadás



- Implicit típusmegadás: a változó típusa az adattartalomából (az inicializáló értékből) következik
- Explicit típusmegadás: a változó típusát rögzítjük a forrásban, nem az inicializálásból derül ki.

```
let firstName = 'Masterfield';  
  
let lastName: string = 'Oktatóközpont';
```

TypeScript függvények



- Az alprogramok készítése során meghatározzuk
 - a formális paraméterek típusát (ha nem állítjuk be automatikusan *any* típust vesznek fel)
 - a függvény visszatérési típusát
- Az overloading részlegesen megvalósítható.
 - Több függvényoszignatúra lehet, de csak egy implementáció

```
function add(a:string, b:string) : string;  
function add(a:number, b:number) : number;  
function add(a: any, b: any): any {  
    return a + b;  
}
```

implementáció

overloading

Paraméterek kezelése



- Megadhatunk opcionális paramétereket (?).

```
function multiplication(a : number, b? : number) : number {  
  let op2 = b || a;  
  return a * op2;  
}
```

- Adhatunk meg alapértelmezett paraméterértékek.

```
function multiplication(a : number, b : number = a) : number {  
  return a * b;  
}
```

- Az utolsó paraméter lehet *Rest* paraméter, amellyel kezelhető a változó paraméterlista.

```
function multiplication(...numbers) : number {  
  let sum = 1;  
  for(let number of numbers)  
    sum *= number;  
  return sum;  
}
```

Type alias



- A függvénytípusok segítségével szétválasztható a függvény típusleírása és implementációja.
- A függvénytípusban meghatározzuk
 - a függvény visszatérési típusát,
 - a paraméterek típusát
- Nem azonos a *fat arrow* függvénymegadással

```
type Predicate = (number : number) => boolean;  
  
const isNegative : Predicate = (number : number) => number < 0;  
const isOdd : Predicate = (number : number) => number % 2 !== 0;
```

Osztályok



- Az attribútumokat és a kezelő metódusokat egy egységbe zárjuk.
- A láthatóságon keresztül szabályozzuk, hogy ki férhet hozzá az osztály elemeihez:
 - **private**: csak a bevezető osztály látja
 - **protected**: definiáló osztály + leszármazottak
 - **public**: alapértelmezett, mindenki látja
- Konstruktor: felelős a példány inicializálásáért.
- Az osztályon belül minden példányhivatkozásnál használni kell a *this* kulcsszót!

Interface



- Definiálja az alkalmazások felületét → csak azt mondjuk meg, hogy milyen mezőkkel és metódusokkal kell rendelkezni, implementációt nem írhatunk.
- Csak típusellenőrzésre használjuk, nem készül belőle JavaScript forrás.

```
interface UserProfile {  
    username: string;  
    email: string;  
    age: number;  
    status?: string; // Az 'status' mező opcionális  
  
    get Status();  
    getStatus();  
}
```

Öröklődés, polimorfizmus



- Az öröklődés segítségével meg tudjuk valósítani az osztályok között a kódmegosztást.
 - A gyermekosztály konstruktora muszáj, hogy meghívja az őt konstruktort.
- Öröklődéssel kapcsolatos szabályok
 - Az osztálynak csakis egy ősz osztálya lehet.
 - Az osztály bármennyi interfacet kiterjeszthet
 - Az interface bármennyi interfacet kiterjeszthet, de osztály nem lehet őse.
- A polimorfizmus szabálya miatt a gyermek felülírhatja az őstől megörökölt metódust.

Generikus



- A metódusok, osztályok, interface-k típussal történő paraméterezése.
- Megkötéseket adhatunk arra vonatkozóan, hogy a típusnak milyen elemeket kell kiterjeszteni.
- Általánosabb kód készítése.

generikus

típusmegszorítás

```
class City<T extends IPerson>{  
    people : T[] = [];  
  
    add(person : T){  
        this.people.push(person);  
    }  
}  
...  
  
let city = new City();  
city.add(new Employee());
```

TypeScript telepítése



- CLI: Command Line Interface
- Parancssori eszköz az Angular projektekhez.
- CLI telepítés:

```
npm install -g typescript
```

globális
telepítés

- Windows Power Shell esetén a PS parancsfájlok használatát előzetesen engedélyezni kell:

```
Set-ExecutionPolicy -Scope CurrentUser  
-ExecutionPolicy RemoteSigned
```

TypeScript fordítás



- A TypeScript forrásokat **.ts** kiterjesztésű állományokban tároljuk.
- A böngésző és a Node.js csak JavaScript forrást tud futtatni, ezért a TypeScriptet lefordítjuk.
- ***tsc* <fájl>**: előállítja a fájl **.js** kimenetét fordítással
- Projekt szintjén a ***tsconfig.json*** fájlban konfigurálhatjuk a fordítás lehetőségét.

```
tsc --init
```

Konfigurációs állomány
létrehozása alapbeállításokkal

Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Interaktív UI létrehozása

4. Adatkötés a komponensekben

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine



- Az Angular koncepciója 4 központi elemre épül:
 - **Komponens:** az alkalmazás alapvető építőköve, kapcsolattartás a felhasználóval.
 - **Sablon:** a böngészőben megjelenő HTML tartalmak leírása, futási időben kiértékelendő JavaScript kifejezésekkel
 - **Direktíva:** az alkalmazás futása során a DOM viselkedését tudjuk megváltoztatni
 - **Dependency Injection:** az alkalmazáson belüli függőségek nyilvántartása, beszúrása, kezelése

Komponensek



- Az alkalmazások fő építőelemei, a böngészőben megjelenő nézetet és azok működését írják le.
- Különálló elemek, amelyeket egy **@Component** dekorátor kapcsol egy egységbe.
 - sablon, amely leírja a megjelenést (html)
 - stílus, amely leírja a formázását (css)
 - osztály, amely leírja a viselkedést (ts)
 - selector, amellyel azonosítható a komponens más sablonokban
- A létrehozás során legenerálásra kerül egy tesztelő állomány is.



- Minden komponens esetén kötelező a sablon definíciója!
- A komponens felhasználói felületen történő megjelenését írja elő.
 - HTML: statikus HTML tartalom
 - Komponensek: más komponensek felhelyezése
 - kifejezések: utasítások végrehajtása az adatok megjelenítésére
- A sablonok az oldalak egy-egy részét írják le, így nincsenek `<html>`, `<body>`, stb. elemek.
- A `<script>` taget nem támogatja!

Direktíva



- A DOM elemeket manipulálja megjelenés és viselkedés vonatkozásában.
- Három típusát különböztetjük meg:
 - **Komponens direktíva:** speciális direktíva, sablon csatolása
 - **Szerkezeti direktíva:** DOM elemek hozzáadása, törlése.
 - **Attribútum direktíva:** DOM elemek megjelenése és/vagy viselkedése.
- Saját direktívát is tudunk készíteni a *@Directive* dekorátor használatával.

Dependency Injection (DI)



- A fejlesztés során az újrahasznosítható kódok, funkciók szolgáltatásként valósulnak meg.
- A komponensek működése a szolgáltatásoktól függenek.
- **DI**: osztály a függőségeit külső forrásból kapja, ahelyett, hogy saját maga hozza létre azokat.
- Lazább csatolást, jobb újrahasznosítást biztosít, könnyebb a tesztelés.
- A későbbiekben például az API hívások során fogjuk használni.

Dependency Injection



- A komponenseknek a szolgáltatások által biztosított funkcionalitásra szükség van.
 - ➔ az komponens példányosítja a szolgáltatást.
- Az osztály a függőségeit külső forrásból kapja, ahelyett, hogy saját maga hozza létre azokat.
- Előnyök
 - Újrafelhasználhatóság növelése.
 - Lazább csatolás a komponensosztály és a függősége között.
 - Könnyebb tesztelés és kódfejlesztés

Angular DI Framework

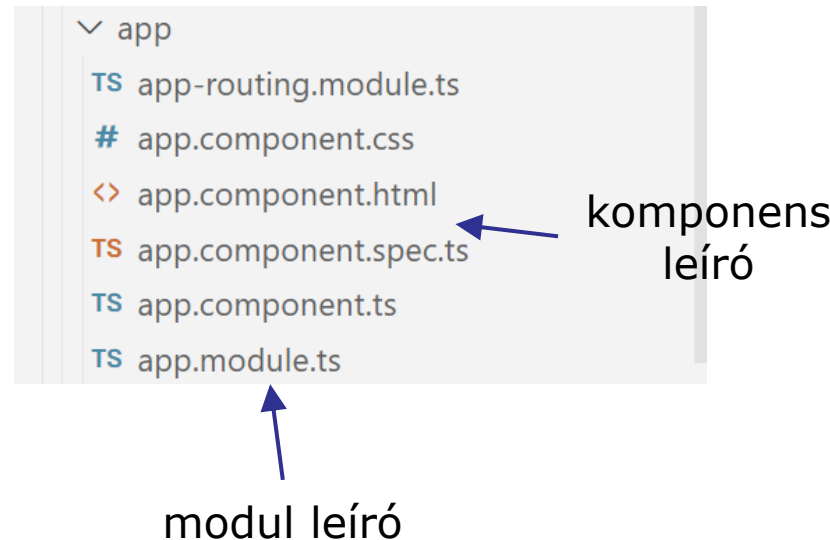
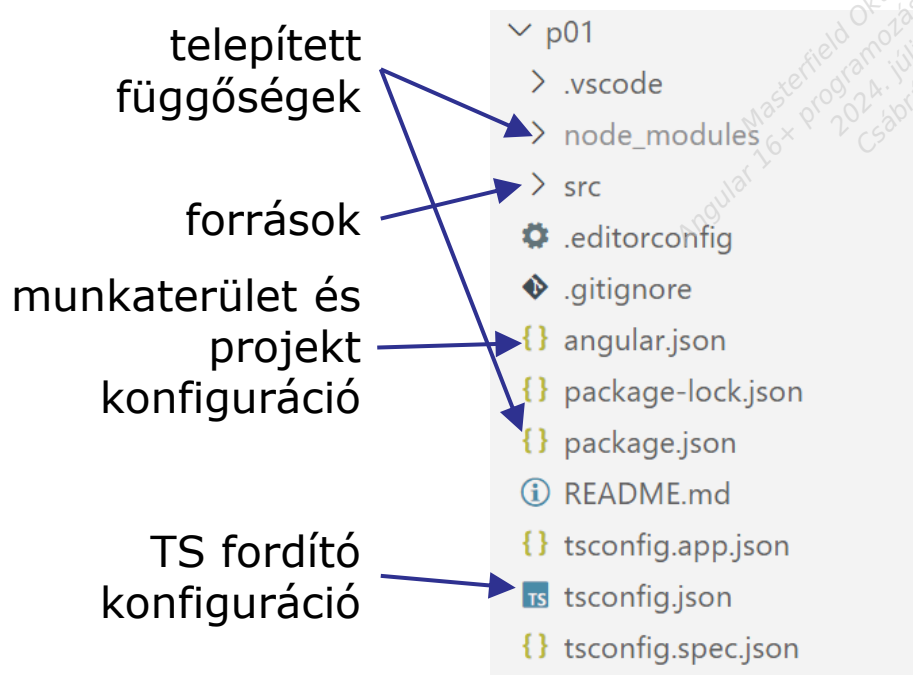


- Létrehozza, karbantartja és beszúrja a függőségeket.
- Elemei:
 - **Consumer:** a függőséget igénylő
 - **Dependency:** a függőség
 - **Injection Token:** a függőség egyedi azonosítója
 - **Provider:** függőségek nyilvántartása
 - **Injector:** felel a függőségek feloldásáért
- Folyamat: az *Injector* megkeresi a függőséget az *Injection Token* alapján, létrehozza, majd beszúrja.

Projektstruktúra



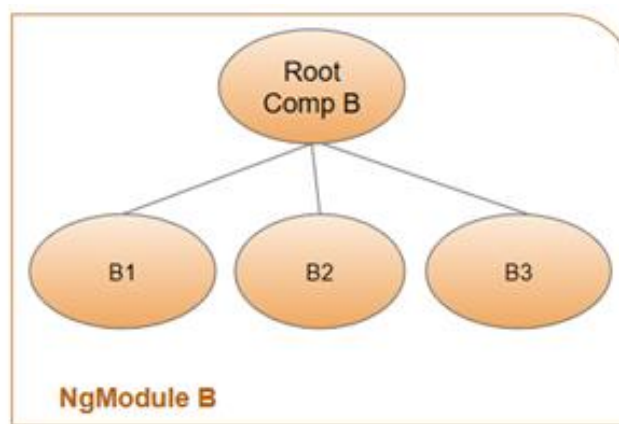
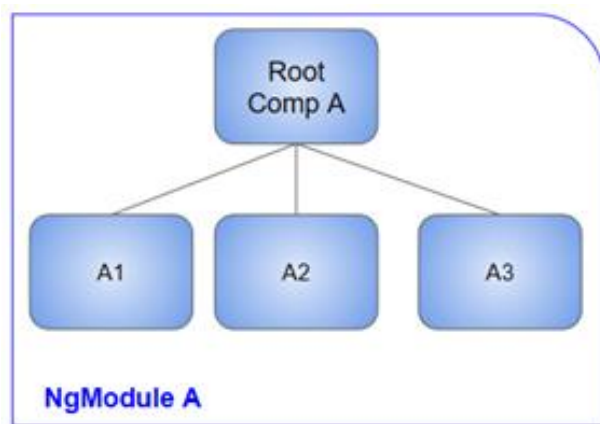
- Workspace (munkaterület): egy vagy több projekt és/vagy könyvtár forrását tartalmazza.
- Projekt: egy-egy önállóan futtatható alkalmazást leíró fájlok összessége.



Projekt struktúrája - modulok



- NgModules: az Angular moduláris rendszere.
- Konténerek egy alkalmazási tartományhoz, egy munkafolyamathoz.
- Importálhatják más modulok exportált elemeit.
- Exportálhatják a kiválasztott funkciókat.
- A gyöker modult **root**-nak nevezték el
- Hierarchikus szerkezetet alkotnak.



Standalone komponensek



- Olyan komponens, amely nem tartozik modulhoz.
- Angular 17-től alapértelmezetten *standalone* komponensek jönnek létre.
- Általában a gyakran használt komponenseknél használjuk a technikát, így elkerülhető az @NgModule direktíva kezelése.
- Angular 17-től a létrehozás pillanatában sem jön létre modul, csak ha azt külön kérjük a *--no-standalone* kapcsolóval.

Angular CLI



- CLI: Command Line Interface
- Parancssori eszköz az Angular projektekhez.
- CLI telepítés:

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

globális
telepítés

- Windows Power Shell esetén a PS parancsfájlok használatát előzetesen engedélyezni kell:

```
Set-ExecutionPolicy -Scope CurrentUser  
-ExecutionPolicy RemoteSigned
```

Parancsok felépítése



- `ng <parancs> [opciók]`
 - `ng`: Angular CLI indítása
 - `parancs`: az Angular CLI által végrehajtandó feladat
 - `opciók`: parancstól függő kiegészítő paraméterek
- Fontosabb parancsok:
 - `ng new`: projekt létrehozása
 - `ng generate`: projektelelem létrehozása
 - `ng test`: tesztek futtatása
 - `ng serve`: fejlesztői szerver indítása
 - `ng build`: projekt buildelése

Angular projekt létrehozása



- Az Angular CLI-n keresztül új projekt készítése:

```
ng new <projekt_neve>
```

- A létrehozás során kapcsolók segítségével konfigurálható a folyamat (ekkor nem kérdez).

```
ng new <kapcsolók> <projekt_neve>
```

- Angular v17-es verziótól alapértelmezés szerint *standalone* projektek jönnek létre. Ha ezt nem szeretnénk, akkor *--no-standalone* kapcsoló.
- Fejlesztői szerver indítása (a projekt mappából):

```
ng serve
```

Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Interaktív UI létrehozása

4. Adatkötés a komponensekben

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine

@Component dekorátor



- Használatához importálnunk kell (@angular/core)
- Közvetlenül az osztálydefiníció előtt írjuk le.
- Fontosabb tulajdonságok
 - selector: a komponens egyedi kiválasztója
 - providers: használt szolgáltatások
 - directives: viselkedést befolyásoló elemek
 - styles/styleUrls: alkalmazott stílusbejegyzések
 - template/templateUrl: a komponens nézete

Komponens osztály



- Alapvető TypeScript osztály.
- Az osztály feladata a komponens működésének leírása:
 - Adathozzáférés: a komponens sablonjában az itt biztosított adatokhoz férünk hozzá. A tárolt adatokat nevezzük állapotnak.
 - Eseménykezelés: segédfüggvények, amelyek az egyes UI eseményekre reagálnak, megváltoztatják az állapotot.

Komponens létrehozás



- Manuálisan
 - A szükséges állományok létrehozása, majd a komponens leírása.
 - A komponenst kézzel adjuk a modulhoz (ha nem standalone)
- Parancssorból (a projekten belül):

```
ng generate component <komp. neve>
```

- Alapértelmezés szerint *standalone*, modul alapú megvalósításnál a parancssori hozzáadás a modulhoz is hozzáadja a komponenst.

Direktíva



- A sablon készítésénél a nézetelemek meghatározásához tudjuk felhasználni.
 - Szerkezeti direktívák: segítségükkel a nézetek készítése során tudjuk alkalmazni a vezérlési szerkezeteket (elágazás, ciklus)
 - Attribútum direktíva: megjelenés/viselkedés meghatározása a megjelenített elemeknél.
- Megjelenés, mint speciális attribútumok.

Direktívák



- A szerkezeti direktívákat * karakterrel kezdjük.
 - ngIf: egyágú elágazás megvalósítása
 - ngSwitch: többágú elágazás megvalósítása
 - ngFor: tömbök bejárása a nézetkészítés során
- Az attribútum direktíva nevét [] közé tesszük.
 - ngClass: az elem dinamikus CSS osztálya
 - ngStyle: az elem dinamikus inline stílusa

```
<table *ngIf="exams.length !== 0">
  ...
  <tr *ngFor="let exam of exams">
    <td>{{exam.student_id}}</td>
    <td>{{exam.result}}%</td>
  </tr>
  ...
</table>
```

Pipeok



- Az adatok formázott megjelenítésére használjuk a megjelenítés során (pl. dátumok formátuma).

```
kifejezés | pipe[:pipe paraméterek]
```

- A pipeok egymás után láncolhatóak: az egyik pipe kimenete lehet a következő bemenete.
- Számos beépített pipe-ot használhatunk a megjelenítés során: currency, date, number, percent, decimal, slice stb.

Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Adatkötés a komponensekben

4. Interaktív UI létrehozása

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine



- Adatkötés/események kötése a felhasználói felület és a modell elemei között történik.
- A kötés célja, hogy szinkronban tartsa a felhasználói felületet és a komponenst.
- Adatkötés típusai:
 - **Egyirányú adatkötés:** a szinkronizáció csak egy irányban történik (nézet → komponens vagy komponens → nézet)
 - **Kétirányú adatkötés:** az adatkötés oda-vissza megvalósul a nézet és a komponens között

Szöveg interpoláció



- A kifejezés kiértékelését követően az eredményt szöveggként helyettesíti be. Ha a sablonkifejezés értéke változik, a nézet frissül.
- A komponens elemeire hivatkozhatunk.
- Jelölése: **{{sablonkifejezés}}**

```
export class BindingsComponent {  
  name = 'Masterfield';  
}
```

```
<p>Hello {{name}}!</p>
```

Tulajdonság kötés



- A HTML elem tulajdonságát a komponens egy tulajdonságához kötjük.
- A forrás lehet tulajdonság, metódus, sablon referenciaváltozó vagy ezekből épített kifejezés.
- Jelölése: **[tulajdonság]="kötés forrása"**

```
export class BindingsComponent {  
  name = 'Masterfield';  
  disabled = false;  
}
```

```
<input [disabled]="disabled" value="name"/>
```

Attribútum kötés



- Bizonyos esetekben nincs lehetőség tulajdonság kötésre, ilyenkor közvetlenül az attribútumhoz kötünk értéket (pl. képernyőfelolvasó szoftver attribútumok).
- Jelölése: **[attr.név]="kötés forrása"**

```
export class BindingsComponent {  
  buttonText = 'Masterfield';  
  disabled = false;  
}
```

```
<button [disabled]="disabled" [attr.aria-label]="buttonText"/>
```

Esemény kötés



- A sablonban bekövetkező eseményekhez a komponens valamely metódusát rendeljük hozzá.
- Az eseménykezelés megvalósítása Angularban.
- Jelölése: **(esemény)="metódushívás"**

```
export class BindingsComponent {  
  onSave(){  
    console.log('Click save button')  
  }  
}
```

```
<button (click)="onSave()">Save</button>
```

@Input dekorátor



- Biztosítani kell, hogy a komponensek kívülről is kaphassanak adatokat.
 - Programozó által definiált adatok.
 - A szülő komponenstől származó adatok.
- A bejövő adatokat **@Input** dekorátorral jelöljük.
 - required: kötelező értéket adni neki
 - alias: a tulajdonság külső neve

```
...  
export class BusinessCardComponent {  
  @Input({required: true, alias: 'fullName'}) name : string;  
  @Input({required: true}) email : string;  
  @Input() phoneNumber : string;  
}
```

@Output dekorátor



- Segítségével a két irányú adatáramlás valósítható meg, mivel az @Output dekorátorral megjelölt mező változásiára a szülő iratkozhat fel.
- Tulajdonképpen eseményfeliratkozás.
- A kimenő adatokat **@Output** dekorátorral jelöljük.

```
@Component({
  selector: 'app-child',
  template: `<button (click)="sendMessage()">Send Message</button>`
})
export class ChildComponent {
  @Output() messageEvent = new EventEmitter<string>();

  sendMessage() {
    this.messageEvent.emit('Hello from Child');
  }
}
```


Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Adatkötés a komponensekben

4. Interaktív UI létrehozása

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine

Űrlapok



- Űrlapok segítségével támogatjuk a felhasználói adatbevitelt. → Az adatokban nem bízunk meg, azokat validálni kell.
 - Az űrlapkezelés során képesnek kell lenni az adatok kezelésére és validálására.
- Angular űrlapkezelési megközelítés:
 - model vezérelt/reaktív (model-driven): közvetlen hozzáférés az űrlapobjektum modellhez
 - sablon vezérelt (template-driven): az űrlap logikája az űrlap sablonjában kerül implementálásra

Sablon vezérelt vs model vezérelt űrlapok



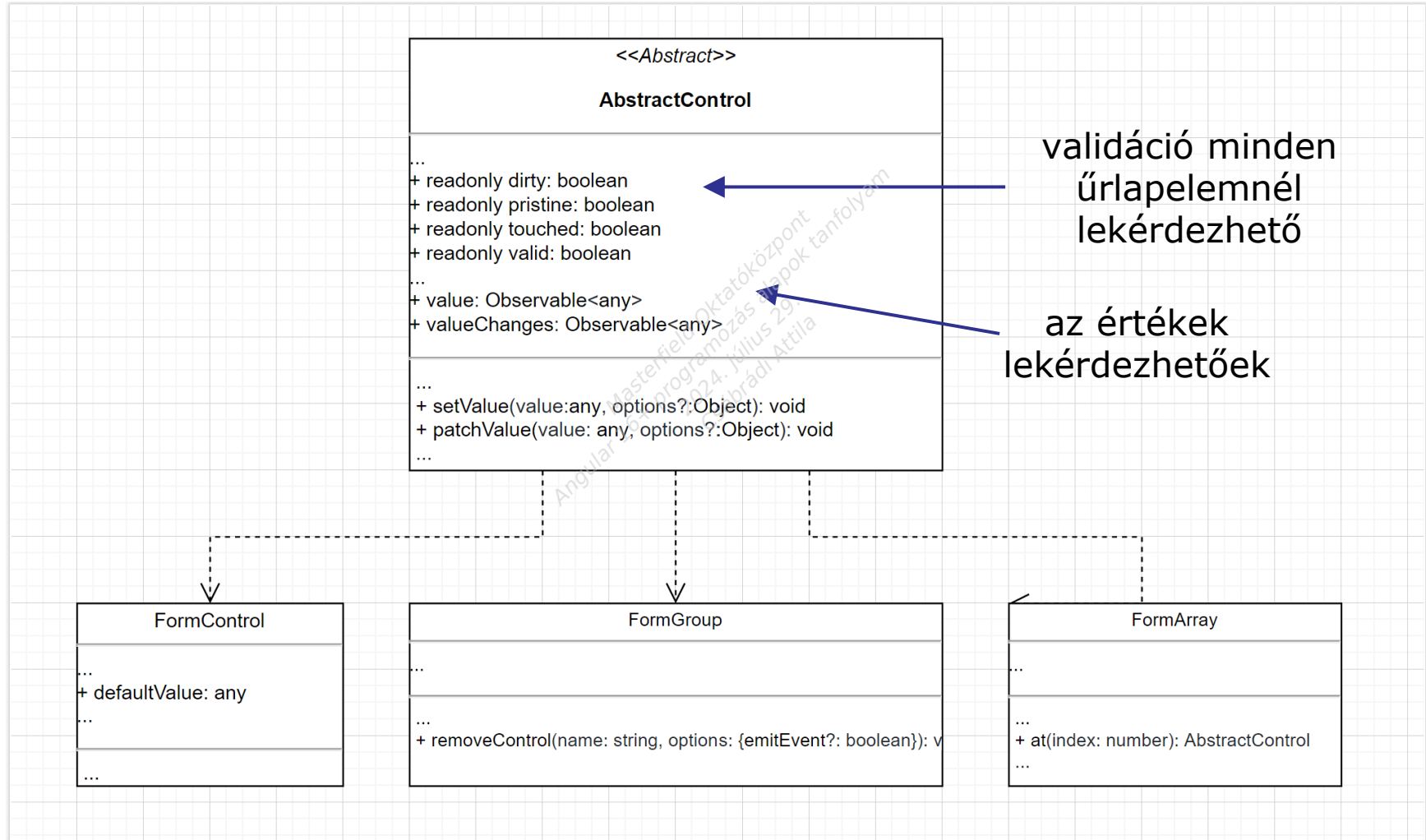
	Sablon vezérelt űrlap	Model vezérelt űrlap
Űrlapmodell	implicit, direktívákkal leírva	explicit, modellben leírva
Adatmodell	strukturálhatatlan és módosítható	strukturált és módosíthatatlan
Adatáramlás	aszinkron	szinkron
Validáció	direktíva alapú	függvényalapú

Űrlapok reprezentálása



- 1. AbstractControl:** űrlapelem közös őse, biztosítja a közös felületet.
- 2. FormControl:** egy egyszerű beviteli mező, a hozzá kapcsolódó összes adatával.
- 3. FormGroup:** FormControl típusú elemek gyűjteménye, egységes kezelést biztosít. Támogatja a beágyazást (belső csoport vagy tömb).
- 4. FormArray:** FormControl elemek tömbje, hasonló a FormGrouphoz, de tömbként reprezentálja a gyűjteményt. Támogatja a beágyazást (belső csoport vagy tömb). Dinamikus űrlapkészítésnél használjuk.

Űrlapok reprezentálása



Sablon vezérelt űrlapok



- Egyszerű űrlapkészítési technika.
- Az űrlap modell implicit módon kerül a komponens osztályba.
- A sablonban az ngModel direktíva hozza létre és kezeli a FormControl példányt az adott elem esetén.
- Közvetett kapcsolat a FormControl példánnyal az ngModel-en keresztül.
- A sablonban leírt direktívák és attribútumok segítségével határozzuk meg a viselkedést és validációt.

Sablon vezérelt űrlap példa



A sablonvezérlés űrlap létrehozza az ngForm-ot. Mi elnevezzük nézetváltozóval

Submit eseményre átadjuk az űrlapot (FormGroup példány).

```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">
  <label for="name">Name</label>
  <input type="text" id="name" name="name" ngModel>

  <label for="age">Age</label>
  <input type="number" id="age" name="age" ngModel>

  <button type="submit">Submit</button>
</form>
```

Létrehozzuk az FormControl
elemeket az űrlapon

Reaktív űrlapok



- Az űrlap modell explicit módon a komponens osztályba kerül.
- A sablonban a `[formControl]` direktíva közvetlenül hivatkozza az űrlap modellt.
- Közvetlen kapcsolat a FormControl példánnyal, folyamatosan kezeljük a változásokat.

Reaktív űrlapok



- Az űrlaphoz kapcsolódó elemeket és validációs szabályokat a komponens osztályban definiáljuk. Majd kötjük a template-hez az űrlapot.
- Lépések:
 1. ReactiveFormsModule importálása
 2. Form Model készítése a komponens osztályban
 3. Űrlap elkészítése
 4. Űrlap kötése a Form Modelhez

Űrlap helyességének ellenőrzése



- A FormControl valid mezőjén keresztül, ugyanez lekérdezhető az űrlaptól is.
- Lehetséges állapotok:
 - **valid**: volt-e hiba, anull érték érték valid
 - **dirty**: a felületen keresztül a felhasználó módosított
 - **touched**: a felületen történő műveletek miatt lefutott az űrlapelemre a blur esemény

Reaktív űrlap példa



Létrehozunk a
FormGroup-ot a
FormControl elemekkel.

```
export class FormComponent {  
  contactForm = new FormGroup({  
    name: new FormControl(),  
    age: new FormControl()  
  })  
  
  submit(){  
    console.log(this.contactForm)  
  }  
}
```

Összerendeljük a modell
elemeit a nézetekkel.

```
<form [formGroup]="contactForm" onSubmit="submit()">  
  <label for="name">Name</label>  
  <input type="text" id="name" name="name" formControlName="name">  
  
  <label for="age">Age</label>  
  <input type="number" id="age" name="age" formControlName="age">  
  
  <button type="submit">Submit</button>  
</form>
```

Validáció



- A felhasználó által megadott adatok ellenőrzése.
- A Validation direktíván keresztül valósítható meg.
- A Forms Module beépített validátorai:
 - required: kötelező kitölteni
 - minlength: minimális karakterhossz
 - maxlength: maximális karakterhossz
 - pattern: reguláris kifejezés alapú minta
 - email: e-mail cím

Validáció használata



```
<form #contactForm="ngForm" (ngSubmit)="onSubmit(contactForm)">
  <label for="name">Name</label>
  <input type="text" id="name" name="name" required>

  <label for="age">Age</label>
  <input type="number" id="age" name="age" min="0" max="100">

  <button type="submit">Submit</button>
</form>
```

Sablon-vezérlés űrlap esetén a sablon határozza meg a validátorokat

```
export class FormComponent {
  contactForm = new FormGroup({
    name: new FormControl('', [Validators.required]),
    age: new FormControl('', [Validators.min(0), Validators.max(100)])
  })
}
```

Reaktív űrlap esetén a modell helyezi fel a validátort

Saját validátor készítése



- Saját Angular direktívát kell készíteni, amely megvalósítja a Validator interfacet.
- validate metódus:
 - paraméterben kap egy FormControl-t
 - null-t ad vissza, ha az adat valid
 - Hibát ad vissza, ha az adat nem helyes

```
interface Validator {  
  validate(control: AbstractControl): ValidationErrors | null  
  registerOnValidatorChange(fn: () => void)? : void  
}
```

Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Adatkötés a komponensekben

4. Interaktív UI létrehozása

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine

Forgalomirányítás célja



- Az SPA célja, hogy az oldalon megjelenő nézetet és tartalmat dinamikusan frissítsük az oldal teljes újratöltése nélkül.
- Az **Angular Router** célja, hogy a nézetváltásokat könnyen és egyszerűen tudjuk leírni.
- Az útválasztó biztosítja a navigációt úgy, hogy a böngésző URL-jét a nézet megváltoztatására vonatkozó utasításként értelmezi.
- Az Angular Router alapértelmezetten telepítésre kerül a projekt létrehozásakor.

Angular Router elemei



- Az @angular/router könyvtárban definiált Angular Router segítségével történik az útvonal alapú navigáció.
 - URL alapú navigáció a nézetekhez
 - URL alapú paraméterek átadása a nézetnek
 - Dinamikus nézetbetöltés
 - Útvonalak védelme illetéktelen felhasználóktól
 - Köti a linkeket a nézethez

A forgalomirányítás elemei



- **Router:** szolgáltatás, amely megvalósítja a navigációt
- **Route:** objektum az útvonal és komponens összerendelésére
- **Routes:** útvonalak gyűjteménye
- **RouterOutlet:** direktíva, a tartalombetöltés helye
- **RouterLink:** direktíva, a HTML elem és az útvonal kötése.
- **RouteLink Parameters tömb:** útvonal paraméterei vagy argumentumai.

Forgalomirányítás folyamata



1. Az oldal alap címének beállítása

```
<base href="/">
```

2. Útvonalak készítése

```
const appRoutes = [{ path: 'product', component: ProductComponent } ]
```

3. Útvonalak regisztrálásra a Router számára

```
import { RouterModule } from '@angular/router';  
...  
imports: [RouterModule.forRoot(appRoutes)],
```

4. Útvonalak megjelenítése

```
<a [routerLink]="['product']">Product</a>
```

5. Komponens betöltése útvonal alapján

```
<router-outlet></router-outlet>
```

Útvonalak



- Útvonalválasztás esetén az első illeszkedő útvonalat fogja használni az Angular → fontos az útvonalak sorrendje.

```
export const appRoutes: Routes = [  
  { path: 'home', component: HomeComponent },  
  { path: 'contact', component: ContactComponent },  
  { path: 'product', component: ProductComponent },  
  { path: '', redirectTo: 'home', pathMatch: 'full' },  
  { path: '**', component: ErrorComponent }  
];
```

Hibás webcím
megadása esetén a
hiba oldalt töltjük be

Alapértelmezett
útvonal esetén
átirányítás

Angular Route Guards



- Az alkalmazás működése során bizonyos oldalakhoz, csak feltételek esetén férhetünk hozzá.
- A beépített Guard az alábbi esetekben használható
 - navigáció megerősítése
 - hozzáférés csak adott felhasználóknak
 - navigáció előtt paraméterek ellenőrzése az útvonalban
 - adatkiolvasása a megjelenítés előtt

Elérhető guardok



- Elérhető guardok listája
 - **CanActivate:** előfeltétel ellenőrzése a navigáció megkezdése előtt.
 - **CanDeactivate:** előfeltétel ellenőrzése a másik megkezdése előtt.
 - **Resolve:** késleltetett navigáció
 - **CanLoad:** megakadályozza a Lazy Loaded modulok betöltését
 - **CanActivateChild:** előfeltétel ellenőrzése a gyermek útvonal eléréséhez

Használat



1. Saját guard elkészítése szolgáltatásként a megfelelő interface implementálásával.

```
import { CanActivate } from '@angular/router';  
@Injectable()  
export class ProductGuardService implements CanActivate {}
```

2. Guard metódus implementálása

```
canActivate(): boolean {  
    // logikai érték, hogy aktiválható-e az útvonal  
    return true;  
}
```

3. Guard regisztrálása szolgáltatásként

```
providers: [ProductService, ProductGuardService]
```

4. Guard hozzáadása az útvonalhoz.

```
{ path: 'product', component: ProductComponent, canActivate : [ProductGuardService] }
```

Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Adatkötés a komponensekben

4. Interaktív UI létrehozása

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine

Szolgáltatások



- A szolgáltatás egy újrafelhasználható kód egy meghatározott cél végrehajtására.
- Tulajdonképpen az Angular szolgáltatás egyszerűen egy Javascript függvény.
- Minden elemtől független, egy-egy jól körülhatárolható funkcionálisitást ír le.
- Előnyei
 - A szolgáltatások tesztelése egyszerűbb.
 - Könnyebb a hibakeresés.
 - A szolgáltatást bárhol használhatjuk.
- Például naplózás, vagy API hívás.

HttpClient



- A HttpClient egy különálló modell, amely az *@angular/common/http* csomagban érhető el.
- Szolgáltatásként valósul meg, injektálni kell a komponens konstruktorában.

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

@Component({...})
export class UsersComponent {
  constructor(private httpClient :HttpClient){

  }
}
```

HttpClient metódus



- A HttpClient minden http módszerhez kínál egy metódust.
 - Az első paramétere a végpont.
 - A második paraméterben pedig a kiegészítő adatokat adjuk meg (tartalom, fejléc, stb.)
- A metódus végrehajtása aszinkron:
 - Nem azonnal kerül végrehajtásra.
 - Feliratkozhatunk és a kérés végrehajtás után megkapjuk az eredményt.
- Haladó szinten a kapott eredményt van lehetőség a lekérdezés során szűrni és transzformálni.

HttpClient példa – GET kérés



Megfogalmazunk egy
GET kérést az adott
végpontra

```
loadData(){  
  const API_URL = 'https://dummyjson.com/products';  
  
  this.httpClient.get(API_URL)  
    .subscribe({  
    next: (data) => {},  
    error: (error) => {},  
    complete: () => {}  
  });  
}
```

A subscribe-al elindítjuk a
kérést és megfogalmazzuk
a feldolgozó függvényeket

next(): megérkezett az adat
error(): hiba keletkezett a végrehajtás során
complete(): befejeződött a feliratkozás

HttpClient példa – POST kérés



Létrehoztam a paramétereket és beépítjük a kérésbe

```
sendData(){  
  const API_URL = 'https://dummyjson.com/products';  
  const headers = new HttpHeaders({  
    'Content-Type': 'application/json'  
  });  
  
  this.httpClient.post(API_URL, {title: 'new product'}, {headers})  
    .subscribe({  
      next: (data) => {},  
      error: (error) => {},  
      complete: () => {}  
    });  
}
```

A subscribe-al elindítjuk a kérést és megfogalmazzuk a feldolgozó függvényeket

A POST tárgya, a küldött adat



- Az Angular az RxJs könyvtárat használja a megfigyelhető elemek megvalósításához.
- Lehetővé teszi a kérések és válaszok átalakítását, feldolgozását és kezelését.
- Az RxJS pipe-ok és operátorok segítségével ezeket a megfigyelőn keletkező válaszokat feldolgozhatjuk különböző módokon.
 - pipe(): transzformációt leíró metódussor
 - map(): átalakítás megvalósítása
 - filter(): szűrés megvalósítása
 - catchError(): hibakezelés a folyamat során

RxJS példa



```
getUsers(): Observable<string[]> {  
  return this.http.get('https://jsonplaceholder.typicode.com/users')  
    .pipe(  
      // Hiba esetén egy üres tömböt adunk vissza  
      catchError(error => of([])),  
      // A JSON-t tömbbé alakítjuk  
      map((data: any[]) => data.map(user => user)),  
      // Csak a 18 év felettieket szűrjük  
      filter(user => user.age >= 18),  
      // A felhasználóneveket kisbetűssé alakítjuk  
      map(user => user.username.toLowerCase()),  
      // A neveket kiírjuk a konzolra  
      subscribe(names => console.log(names))  
    );  
}
```

Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Adatkötés a komponensekben

4. Interaktív UI létrehozása

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine



- **Állapotkezelési könyvtár.** Segít az alkalmazás állapotának hatékony kezelésében és az adatáramlás irányításában.
 - **Központi állapotkezelés:** az állapotok egyetlen központi állapotfában kerülnek tárolásra.
 - **Egyértelmű adatfolyam:** reaktív, funkcionális megközelítés az állapotkezelésre.
 - **Aszinkron műveletek kezelése:** kezeli az aszinkron műveleteket az effektusokon keresztül
 - **Skálázhatóság és karbantarthatóság:** független állapotkezelési rendszert biztosít



- Az NgRx állapotkezelés 4 elemre épül:
 - **State:** alkalmazás aktuális állapotát leíró objektum
 - **Actions:** alkalmazásban történő változásokat leíró egyszerű objektumok
 - **Reducers:** függvények, amelyek reagálnak az akciókra, és módosítják az állapotot.
 - **Effects:** aszinkron műveleteket elvégző folyamatok.
 - **Selectors:** az állapotok lekérést célzó függvények.

Állapotkezelés folyamata



1. Állapotok definiálása.

1. Meghatározzuk az állapot tartalmát (AppState)

```
interface Task {  
  id: number;  
  name: string;  
  completed: boolean;  
}
```

```
interface AppState {  
  tasks: Task[];  
}
```

2. Beállítjuk a kezdőértékeket (ha vannak).

```
const initialState: AppState = {  
  tasks: [  
    { id: 1, name: 'Bevásárlás', completed: false },  
    { id: 2, name: 'Megbeszélés', completed: true },  
    { id: 3, name: 'Edzés', completed: false }  
  ]  
};
```

Akciók meghatározása



2. Az akciók leírják az alkalmazásban történő változásokat.

- Egyszerű objektumok, amelyek tartalmazzák a változások típusát és az esetlegesen szükséges adatokat.

```
enum TaskActionTypes {  
    ADD_TASK = '[Task] Add Task',  
}  
  
class AddTask implements Action {  
    readonly type = TaskActionTypes.ADD_TASK;  
    constructor(public payload: Task) {}  
}  
  
type TaskActions = AddTask;
```

Akció típusa

Implementálja az Actiont

Adat

Reducers meghatározása



3. Állapotátmenet leírása: a reducer megkapja az állapotot és az akciót és visszaadja az új állapotot.
- Az állapot vizsgálatánál általában switch-et használunk.

```
export function taskReducer(state: Task[] = [], action: TaskActions)
: Task[] {
    switch (action.type) {
        case TaskActionTypes.ADD_TASK:
            return [...state, action.payload];
        default:
            return state;
    }
}
```

Store kialakítása



4. Az akciók és reducerek összerendelése

1. Az összes reducer leképezést leírjuk

```
const reducers: ActionReducerMap<AppState> = {  
  tasks: taskReducer  
};
```

2. Store létrehozása az előzőből

```
export const appStore = StoreModule.forRoot(reducers);
```

5. Használjuk a Store-t az alkalmazásban

Agenda



1. Bevezetés

2. Angular alkalmazások felépítése

3. Adatkötés a komponensekben

4. Interaktív UI létrehozása

5. Űrlapok készítése

6. Forgalomirányítás és navigáció

7. REST szolgáltatás hívása

8. NgRx

9. Unit testing - Jasmine

Tesztelés



- Biztosítja a szoftver minőségét és stabilitását.
- Ellenőrzi a kód helyes működését, és feltárja az esetleges hibákat vagy mellékhatásokat.
- Tesztelés típusai:
 - **Funkcionális tesztelés** (Functional testing): a rendszer funkcióit teszteljük, azaz azt ellenőrizzük, hogy a szoftver a felhasználó által elvárt módon működik-e.
 - **Elfogadási tesztelés** (Acceptance testing): a rendszert teszteljük az üzleti követelmények alapján.

Tesztelés szintjei az Angularban



- 1. Egységtesztek:** komponensek, szolgáltatások funkcióinak izolált vizsgálata.
 - Eszközök: *Jasmine* és *Karma* kombinációja
- 2. Integrációs tesztek:** az egyes komponensek közötti interakciók és integrációjának vizsgálata
 - Eszközök: *Angular Testing Library* vagy a *Jasmine* és *Karma* kombinációja
- 3. End-to-End tesztek:** teljes működés szimulálása az interakcióktól kezdve a kommunikációig.
 - Eszközök: pl. *Protractor*, *Cypress*

Egységtesztelés célja



- **Kódminőség biztosítása:** ellenőrizhető, hogy a kód megfelel az elvárásnak és a specifikációnak.
- **Stabilitás növelése:** korai fázisban észrevehetőek és kijavíthatók az esetleges hibák.
- **Visszajelzés biztosítása a kód változásaihoz:** módosításkor automatikusan futtatásra kerülnek, így azonnal észrevehetővé válnak a hibák vagy az esetleges mellékhatások.
- **Dokumentáció:** egyfajta dokumentációt is nyújtanak a kódhoz, mivel leírják a funkciók működését.
- **Bizalom növelése:** a tesztek garantálják a kód helyes működését.

Jasmine



- Nyílt forráskódú, tesztkerendszer JavaScripthez.
- Behavior-Driven Development (BDD) stílusú tesztek megvalósítása
- Egységtesztek hatékony megvalósítása.
- Egyszerű és intuitív szintaxissal rendelkezik.
- Támogatja az összes szükséges funkcionalitást az egységtesztek készítéséhez.
- Minden tesztet külön-külön lehet futtatni, ami lehetővé teszi a gyors és hatékony tesztelést.
- Könnyen integrálható a meglévő fejlesztési folyamatokba.
- Gazdag közösség és támogatás.

Behavior-Driven Development (BDD)



- A BDD fő célja a szoftver viselkedésének leírása és megértése az üzleti szempontból.
- A BDD azt hangsúlyozza, hogy a szoftver működését az üzleti követelményeknek megfelelően kell leírni, és ezt a viselkedést tesztekkel kell ellenőrizni.
- A tesztek az üzleti scénáriókra épülnek, és az üzleti célokra fókuszálnak.
- A BDD tesztek általában magasabb szintűek és átfogóbbak lehetnek, mint a TDD (Test-Driven Development) tesztek, mivel a különböző üzleti scénáriókra összpontosítanak.



- Tesztvezérlő keretrendszer, amely lehetővé teszi a tesztek futtatását különböző böngészőkben és környezetekben.
- A Jasmine tesztek futtathatók a Karma segítségével a különböző böngészőkben.
- A Karma lehetővé teszi a tesztelési környezet konfigurálását, beleértve a teszt fájlok és forrásfájlok betöltését, valamint a böngészők beállítását.
- Általában a Jasmine teszteket Karma vezérlővel használjuk.

Tesztelés előkészítése



1. Karma telepítése a projektbe

```
npm install karma karma-jasmine  
karma-chrome-launcher --save-dev
```

2. Jasmine telepítése a projektbe

```
npm install jasmine-core --save-dev
```

3. Tesztek elkészítése

4. Tesztek futtatása (teszt szerver indítása)

```
ng test
```


- célszerű *--code-coverage* kapcsolót használni a lefedettség méréséhez vagy külön config fájlt használni

Karma & Jasmine riport kimenet



Karma v 6.4.3 - connected; test: complete; DEBUG

Chrome 124.0.0.0 (Windows 10) is idle

 **Jasmine** 4.6.0 Options

3 specs, 0 failures, randomized with seed 85901 finished in 0.098s

AppComponent

- should create the app
- should render title
- should have as title 'proj01'

==== Coverage summary =====

Statements : 100% (6/6)

Branches : 100% (0/0)

Functions : 100% (3/3)

Lines : 100% (5/5)

=====

Tesztek készítése



- A tesztelési fájlok kiterjesztése **.spec.ts**, így kiterjesztés alapján azonosíthatóak a tesztfájlok.
- Az egységtesztek az egységek mellett találhatóak közvetlenül, nevük ugyanaz, csak kiterjesztésben térnek el.
- A tesztfájlokban írjuk le a konkrét funkcionalitás tesztjeit:
 - *describe*: a tesztek csoportja
 - *it*: egy-egy független teszteset felvétele
 - *expect* hívás és a *matcherek* használatával írható le a visszatérési érték vizsgálata

Példa



```
describe('MathOperationService', () => {  
  let service: MathOperationService;  
  
  beforeEach(() => {  
    TestBed.configureTestingModule({});  
    service = TestBed.inject(MathOperationService);  
  });  
  
  it('call with two positiveinteger', () => {  
    let p1 = 12;  
    let p2 = 18;  
    let result = 30;  
    expect(service.addition(p1,p2)).toEqual(result);  
  });  
});
```

tesztcsoport

teszteset

vizsgálat

Matchers



- A futási eredményeket ellenőrzik.
 - `toBe()`, `toEqual()`, `not.toBe()`
 - `toBeDefined()`, `toBeUndefined()`, `toBeNull()`, `toBeNaN()`, `toBeFalsy()`, `toBeInstanceOf()`, `toBeTruthy()`
 - `toThrow()`, `toThrowError()`
 - `toBeCloseTo()`, `toBeGreaterThan()`, `toBeLessThan()`, `toBeGreaterThanOrEqual()`, `toBeLessThanOrEqual()`
 - `toMatch()`
 - `toContain()`

Triggerelt események



- Sok esetben a teszteket elő kell készíteni, és a teszt után kiegészítő műveleteket kell végrehajtani. → Triggerelt függvényekben végezhetjük el ezeket a feladatokat.
 - `beforeAll()`: olyan feladat, amely a tesztcsoport összes tesztje előtt 1x fut le.
 - `beforeEach()`: minden teszt előtt lefutó feladat
 - `afterEach()`: mindent teszt után lefutó feladat
 - `afterAll()`: olyan feladat, amely a tesztcsoport összes tesztje után 1x fut le.

És végül...

... kérem tegyék fel kérdéseiket!





**Köszönjük, hogy meghallgatták
előadásunkat!**



**További tanfolyamok és információ:
www.masterfield.hu**