https://github.com/sudheerj/angular-interview-questions

Mi az az Angular?

Az Angular egy TypeScript-alapú nyílt forráskódú front-end platform, amely megkönnyíti az alkalmazások webes / mobilos használatával történő felépítését. Ennek a keretrendszernek a főbb jellemzőit, például a declarative templates, dependency injection és még sok más funkciót használnak a fejlesztés megkönnyítésére.

Mi a TypeScript?

A TypeScript egy tipizált JavaScript-készlet, amelyet a Microsoft hozott létre, amely opcionális típusokat, osztályokat, async / await és sok más funkciót ad hozzá, és fordít az egyszerű JavaScript-hez. Globálisan telepítheti.

```
npm install -g typescript
```

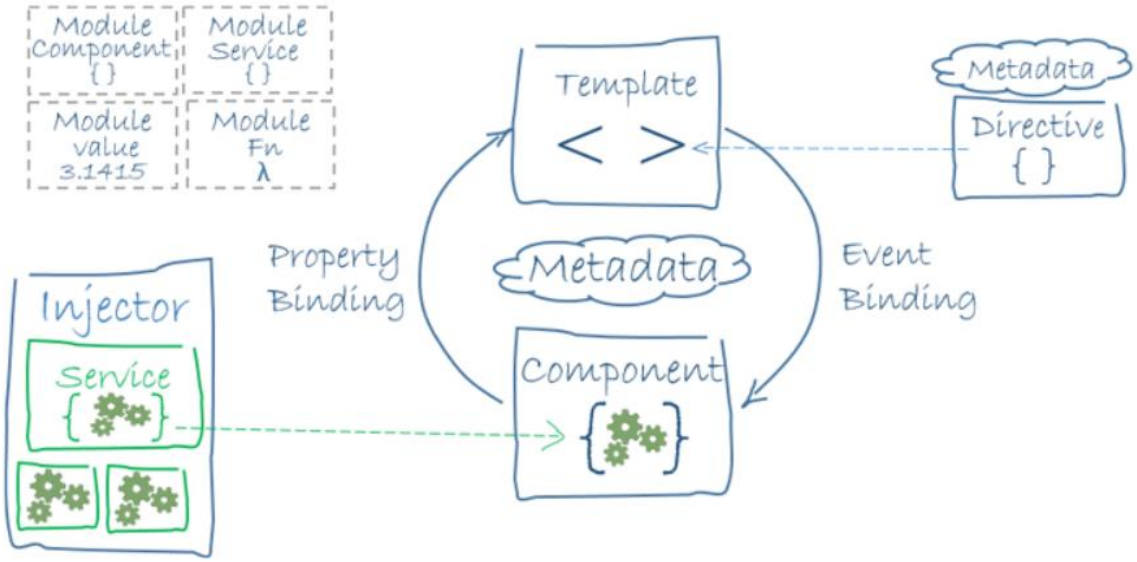
Példa:

```
function greeter(person: string) {
    return "Hello, " + person;
}

let user = "Sudheer";

document.body.innerHTML = greeter(user);
```

Mi az Angular felépítése?



Melyek az Angular legfontosabb elemei?

Az Angular az alábbi kulcsfontosságú elemekkel rendelkezik,

- Komponens: Ezek a HTML nézetek vezérlésének alapvető építőkövei.
- Modulok: A modulok olyan építőelemekből állnak, mint component, directives, services stb.
- Sablonok (Templates): Ez egy Angular alkalmazás nézeteit képviseli.
- Szolgáltatások (Services): Olyan komponensek létrehozására szolgál, amelyek megoszthatók az egész alkalmazásban.
- Metaadatok: Ezzel további adatok adhatók hozzá egy Angular osztályhoz.

Mik azok a direktívák?

A direktívák viselkedést adnak egy meglévő DOM elemhez vagy egy meglévő komponens példányhoz.

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

@Directive({ selector: '[myHighlight]' })
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'yellow';
    }
}
```

Használata:

```
<p myHighlight>Highlight me!</p>
```

Mik a komponensek?

A komponensek a felhasználói felület építőkövei egy Angular alkalmazás során, amely egy Angular komponensekből álló fát alkotott. Ezek az elemek a direktívák részhalmazát képezik. A direktíváktól eltérően a komponenseknek mindig van sablonjuk (template), és egy elemenként a template-ben csak egy komponenst példányosíthatunk. Lássunk egy egyszerű példát a komponensre.

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

@Component ({
   selector: 'my-app',
   template: ` <div>
      <h1>{{title}}</h1>
      <div>Learn Angular6 with examples</div>
   </div> `,
})

export class AppComponent {
   title: string = 'Welcome to Angular world';
}
```

Mi a különbség a component és a directive között?

Röviden: Az component egy template-el rendelkező directive.

Néhány fő különbséget táblázatos formában említünk

Component	Directive
regisztrálásához a @Component meta-adatok	Regisztrálásához @Directive meta-adat annotációt

Component	Directive
annotációját használjuk	használunk
általában felhasználói felület widgetek létrehozására használják	A direktíva a viselkedés hozzáadásához használható egy meglévő DOM elemhez
A komponens az alkalmazás kisebb összetevőkre bontására szolgál	Újrafelhasználható komponensek tervezésére szolgál
DOM elemenként csak egy komponens lehet jelen	Számos direktíva használható DOM elemenként
A @View dekorátor vagy a templateurl / template kötelező	Directive nem használja a View-t

Mi az a sablon (template)?

A sablon (template) egy HTML nézet, ahol az adatokat Angular komponensek tulajdonságainak vezérlőivel történő összekapcsolásával jelenítheti meg. A komponens sablonját a két hely egyikében tárolhatja. Megadhatuk az inline a sablon tulajdonság használatával, vagy meghatározhatjuk a sablont külön HTML fájlban, és a @Component dekorátor templateUrl tulajdonságával összekapcsolhatja a komponens metaadataiban.

Inline sablon használata template szintaxissal,

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

@Component ({
   selector: 'my-app',
   template: '
      <div>
         <h1>{{title}}</h1>
         <div>Learn Angular</div>
      </div>
   '
})

export class AppComponent {
   title: string = 'Hello World';
}
```

Külön sablonfájl, például app.component.html használata

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

@Component ({
   selector: 'my-app',
   templateUrl: 'app/app.component.html'
})

export class AppComponent {
   title: string = 'Hello World';
}
```

Mi az a modul?

A modulok logikai határok az alkalmazásban, és az alkalmazás külön modulokra van osztva az alkalmazás funkcionalitásának elkülönítésére. Vegyünk egy példát az app.module.ts gyökérmodulra, amelyet a @NgModule dekorátorral deklaráltunk az alábbiak szerint

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';

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

Az NgModule dekorator-nak öt fontos lehetősége van (az összes között)

- Az importálás opció más függő modulok importálására szolgál. A BrowserModule alapértelmezés szerint kötelező minden webalapú Angular alkalmazáshoz
- A declarations opcióval definiálhatók az components az adott modulban
- A bootstrap opció megmondja az Angularnak, hogy melyik komponenst (component) indítsa el az alkalmazásban
- A provider opció segítségével konfigurálhatók az injektálható objektumok.
- Az entryComponents opció a nézetbe dinamikusan betöltött összetevők halmaza.

Melyek a lifecycle hook-ok?

Az Angular alkalmazásnak egy teljes életciklusa van, az inicializációtól az alkalmazás végéig.

- ngOnChanges: Ha egy data bound property értéke megváltozik, akkor ezt a módszert hívjuk meg.
- ngOnInit: Ezt hívják, amikor a directive/component inicializálása történik, miután az Angular először megjeleníti az data-bound property-ket.
- ngDoCheck: Ez azoknak a változásoknak az észlelésére szolgál, amelyekre az Angular önmagában nem képes, vagy nem fog érzékelni.

- ngAfterContentInit: Ezt hívjuk meg válaszként, miután az Angular a külső tartalmat a component nézetébe vetíti.
- ngAfterContentChecked: Ezt hívják meg válaszként, miután az Angular ellenőrzi a komponensbe vetített tartalmat.
- ngAfterViewInit: Ezt válaszként hívják meg, miután az Angular inicializálja az component's views és a child views.
- ngAfterViewChecked: Ezt hívják meg válaszként, miután az Angular ellenőrzi az component's views és child views.
- ngOnDestroy: Ez közveltlenül az előtt fut le, hogy az Angular megsemmisítené a directive/component.

Mi az adatkötés?

Az adatkötés az Angular egyik alapkoncepciója, amely lehetővé teszi a komponens és a DOM közötti kommunikáció definiálását, ami nagyon egyszerűvé teszi az interaktív alkalmazások definiálását. Az adatkötésnek négy formája van (3 kategóriára osztva), amelyek eltérnek az adatok áramlásától.

1.) A komponenstől a DOM-ig:

Interpoláció: {{value}}: Egy property értékét hozzáadja a komponensből

```
<li>Name: {{ user.name }}</li>
<li>Address: {{ user.address }}</li>
```

Tulajdonság-kötés (Property binding): [tulajdonság] = ”érték”: Az érték átkerül a komponensből a megadott tulajdonságba vagy egyszerű HTML-attribútumba

```
<input type="email" [value]="user.email">
```

2.) A DOM-tól a komponensig: Eseménykötés (event binding): (esemény) = ”függvény”: Amikor egy adott DOM-esemény történik (pl .: kattintás, változtatás, billentyűzet), hívja meg a komponensben megadott metódust.

```
<button (click)="logout()"></button>
```

3.) Kétirányú kötés (Two-way binding): [(ngModel)] = „érték”: A kétirányú adatkötés lehetővé teszi az adatfolyam mindkét irányba történő lefutását. Például az alábbi kódrészletben mind az e-mail DOM input, mind az komponens e-mail property-je szinkronban van

```
<input type="email" [(ngModel)]="user.email">
```

Mi a metaadat?

A metaadatokat egy osztály dekorálására (decoration) használják, hogy konfigurálhassák az osztály várható viselkedését. A metaadatokat dekorátorok képviselik

1.) Osztály decorator, pl. @Component és @NgModule

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

@Component({
  selector: 'my-component',
  template: '<div>Class decorator</div>',
})
export class MyComponent {
  constructor() {
    console.log('Hey I am a component!');
  }
}

@NgModule({
  imports: [],
  declarations: [],
})
export class MyModule {
  constructor() {
    console.log('Hey I am a module!');
  }
}
```

2.) Property decorators: property-kre használják az osztályon belül, e.g. @Input and @Output

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

@Component({
    selector: 'my-component',
    template: '<div>Property decorator</div>'
})

export class MyComponent {
    @Input()
    title: string;
}
```

3.) Method decorators: Az osztály metódusaira használják, e.g. @HostListener

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

@Component({
    selector: 'my-component',
    template: '<div>Method decorator</div>'
})
export class MyComponent {
    @HostListener('click', ['$event'])
    onHostClick(event: Event) {
        // clicked, `event` available
    }
}
```

4.) Parameter decorators: Osztály konstruktorának paramétereként használjuk, pl. @Inject, Optional

```
import { Component, Inject } from '@angular/core';
import { MyService } from './my-service';

@Component({
    selector: 'my-component',
    template: '<div>Parameter decorator</div>'
})
export class MyComponent {
    constructor(@Inject(MyService) myService) {
        console.log(myService); // MyService
    }
}
```

Mi az az Angular CLI?

Az Angular CLI (Command Line Interface) egy parancssori felület az Angular alkalmazások készítéséhez és felépítéséhez a nodejs stílusú (commonJs) modulok használatával. Az npm paranccsal kell telepíteni,

```
npm install @angular/cli@latest
```

Az alábbiakban felsoroljuk a néhány parancsot, amelyek hasznosak lesznek Angular projektek létrehozása közben

i. **Creating New Project:** ng new

ii. **Generating Components, Directives & Services:** ng generate/g The different types of commands would be,

- ng generate class my-new-class: add a class to your application
- ng generate component my-new-component: add a component to your application
- ng generate directive my-new-directive: add a directive to your application
- ng generate enum my-new-enum: add an enum to your application
- ng generate module my-new-module: add a module to your application
- ng generate pipe my-new-pipe: add a pipe to your application
- ng generate service my-new-service: add a service to your application

iii. **Running the Project:** ng serve

Mi a különbség a konstruktor és az ngOnInit között?

A TypeScript osztályoknak alapértelmezetten van konstruktoruk, amelyet általában az inicializálási célra használnak. Míg az ngOnInit az Angular-ra jellemző, különösen az Angular kötések meghatározására használják. Annak ellenére, hogy a konstruktort először hívják meg, az összes Angular-ral kapcsolatos kötést (binding) célszerű áthelyezni az ngOnInit metódusba. Az ngOnInit használatához az alábbiak szerint kell az OnInit interfészt implementálni,

```
export class App implements OnInit{
  constructor(){
     //called first time before the ngOnInit()
  }

  ngOnInit(){
     //called after the constructor and called  after the first ngOnChanges()
  }
}
```

Mi az a service?

Service-t akkor használnak, amikor közös funkciót kell biztosítani a különféle moduloknak. A service-ek lehetővé teszik az alkalmazással kapcsolatos aggályok nagyobb elkülönítését és a jobb modularitást azáltal, hogy lehetővé teszik az általános funkciók kivonását a komponensekből.

Hozzunk létre egy repoService-t, amely minden komponensben használható:

```
import { Injectable } from '@angular/core';
import { Http } from '@angular/http';

@Injectable({ // The Injectable decorator is required for dependency injection to work
  // providedIn option registers the service with a specific NgModule
  providedIn: 'root',  // This declares the service with the root app (AppModule)
})
export class RepoService{
  constructor(private http: Http){
  }

  fetchAll(){
    return this.http.get('https://api.github.com/repositories');
  }
}
```

A fenti service a Http service-t használja függőségként.

Mi a Dependency Injection Angular-ban?

A Dependency Injection (DI) egy fontos alkalmazás-tervezési minta, amelyben egy osztály külső forrásoktól kéri a függőségeket, ahelyett, hogy azokat maga létrehozná. Az Angular saját függőség-injektálási keretrendszerrel rendelkezik a függőségek (szolgáltatások vagy objektumok, amelyekre az osztálynak a funkciójának ellátásához szükséges) megoldására. Tehát a szolgáltatásai más alkalmazásoktól függenek.

Mi a célja az async pipe-nak?

Az AsyncPipe feliratkozik egy observable-ra vagy promise-ra, és visszaadja a legújabb kibocsátott értéket. Amikor új értéket bocsát ki, a pipe megjelöli az ellenőrizni kívánt komponenst a változások szempontjából.

Vegyünk egy observable-t amely az időt mutatja, amely 2 másodpercenként folyamatosan frissíti a nézetet az aktuális idővel.

```
@Component({
  selector: 'async-observable-pipe',
  template: `<div><code>observable|async</code>:
       Time: {{ time | async }}</div>`
})
export class AsyncObservablePipeComponent {
  time = new Observable(observer =>
    setInterval(() => observer.next(new Date().toString()), 2000)
  );
}
```

Mi a különbség az inline és a external sablonfájl között?

A komponens sablonját a két hely közül az egyikben tárolhatjuk. Megadhatjuk inline a sablon tulajdonság használatával, vagy meghatározhatjuk a sablont külön HTML fájlban, és a @Component dekorátor (decorator) templateUrl tulajdonságával összekapcsolhatjuk.

Az inline és a external HTML közötti választás ízlés, és kód szervezetés kérdése. De általában a kód kis részéhez inline sablont, a nagyobb nézetekhez pedig külső sablonfájlt használunk. Alapértelmezés szerint az Angular CLI template fájlal (external) generál komponenseket. De ezt felülírhatjuk az alábbi paranccsal,

```
ng generate component hero -it
```

Mi a célja az ngFor-nak?

A sablonban az Angular ngFor direktívát használjuk a lista minden elemének megjelenítésére. Például itt ismételjük a felhasználók listáját,

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

Mi a célja az ngIf-nek?

Előfordul, hogy egy alkalmazásnak csak meghatározott körülmények között kell megjelenítenie a nézetet vagy a nézet egy részét. Az Angular ngIf igaz vagy hamis feltétel alapján beszúr vagy eltávolít egy elemet. Vegyünk egy példát egy üzenet megjelenítésére, ha a felhasználó életkora meghaladja a 18 évet,

```
<p *ngIf="user.age > 18">You are not eligible for student pass!</p>
```

Megjegyzés: Az Angular nem jeleníti meg az üzenetet. A bekezdéselem hozzáadása és eltávolítása a DOM-ból. Ez javítja a teljesítményt, különösen a nagyobb adatkötésű nagyobb projektekben.

Mi történik, ha szkript tag-et használunk a sablonon (template) belül?

Az Angular nem biztonságosnak ismeri fel és automatikusan tisztítást csinál, ami eltávolítja a <script> címkét, de biztonságos tartalmat, például a <script> címke szöveges tartalmát megtartja. Így kiküszöböli a szkript-injekciós (script injection) támadások kockázatát. Figyelmeztetés is jelenik meg a böngésző konzolján.

Vegyünk egy példát a innerHtml property binding-ra, amely XSS vulnerability-t okoz,

```
export class InnerHtmlBindingComponent {
  // For example, a user/attacker-controlled value from a URL.
  htmlSnippet = 'Template <script>alert("0wned")</script> <b>Syntax</b>';
}
```

Mi az interpoláció (interpolation)?

Az interpoláció egy speciális szintaxis, amelyet az Angular property binding-ra alakít. Ez kényelmes alternatíva a property binding-ra. Kettős {} zárójel ({{}}) képviseli. A zárójelek közötti szöveg gyakran egy component property neve. Az Angular ezt a nevet a megfelelő component property értékével helyettesíti.

Vegyünk egy példát,

```
<h3>
  {{title}}
  <img src="{{url}}" style="height:30px">
</h3>
```

Mik azok a sablon kifejezések (template expression)?

A sablon kifejezés (template expression) Javascript kifejezéshez hasonló értéket állít elő. Az Angular végrehajtja a kifejezést, és hozzárendeli egy cél tulajdonságához (property of a binding target); ami lehet HTML elem, komponens (component) vagy direktíva (directive). A tulajdonságkötésben (property binding) az alábbit használjuk: [tulajdonság] = "kifejezés". Interpolációs szintaxisban a sablon kifejezést kettős {} zárójel veszi körül. Például az alábbi interpolációban a sablon kifejezés {{ username}},

```
<h3>{{username}}, welcome to Angular</h3>
```

Az alábbi javascript kifejezések tilosak a sablon kifejezésben

- assignments (=, +=, -=, ...)
- new
- chaining expressions with ; or ,
- increment and decrement operators (++ and --)

Mik azok a sablon utasítások (template statements)?

A sablon utasítás (template statements) reagál egy binding target által kiváltott eseményre, például egy elemre, component-re vagy directive-ra. A sablon utasítások (template statement) az alábbi szintaxisban jelennek meg: (event)="statement".

Vegyünk egy példát a gombra kattintás esemény utasítására

```
<button (click)="editProfile()">Edit Profile</button>
```

A fenti kifejezésben az editProfile egy sablon utasítás. Az alábbi JavaScript szintaxis kifejezések nem engedélyezettek.

i. new
ii. increment and decrement operators, ++ and --
iii. operator assignment, such as += and -=
iv. the bitwise operators | and &
v. the template expression operators

Hogyan kategorizálja az adatkötési típusokat (data binding types)?

A kötéstípusok (Binding types) három kategóriába sorolhatók, megkülönböztetve őket az adatáramlás irányától:,

- A forrástól a nézetig (From the source-to-view)
- Nézettől a forrásig (From view-to-source)
- Nézetből forrásba és nézetbe (View-to-source-to-view)

A lehetséges kötési szintaxist az alábbiak szerint lehet táblázatosan felsorolni,

Data direction	Syntax	Type
From the source-to-view(One-way)	1. {{expression}} 2. [target]="expression" 3. bind-target="expression"	Interpolation, Property, Attribute, Class, Style
From view-to-source(One-way)	1. (target)="statement" 2. on-target="statement"	Event
View-to-source-to-view(Two-way)	1. [(target)]="expression" 2. bindon-target="expression"	Two-way

Mik azok a pipe-ok?

A pipe bemenetként veszi fel az adatokat és alakítja át a kívánt kimenetre. Vegyünk például egy pipe-ot, hogy a komponensben a születésnapi dátumot átalakítsuk szebb formátumra.

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

@Component({
  selector: 'app-birthday',
  template: `<p>Birthday is {{ birthday | date }}</p>`
})
export class BirthdayComponent {
  birthday = new Date(1987, 6, 18); // June 18, 1987
}
```

Mi az a paraméterezett pipe?

A pipe tetszőleges számú opcionális paramétert képes elfogadni a kimenet finomhangolásához. A paraméterezett pipe úgy hozható létre, hogy a pipe nevét kettősponttal (:) deklarálja, majd a paraméter értékét. Ha a pipe több paramétert is elfogad, értékeket kettősponttal kell elválasztani. Vegyünk egy születésnapi példát egy adott formátummal (éééé / hh / éééé):

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

    @Component({
      selector: 'app-birthday',
      template: `<p>Birthday is {{ birthday | date:'dd/MM/yyyy'}}</p>` // 18/06/1987
    })
    export class BirthdayComponent {
      birthday = new Date(1987, 6, 18);
    }
```

Hogyan lehet a pipe-okat láncolni?

Vegyünk egy születésnapi tulajdonságot, amely a dátum pipe-ot (a paraméterrel együtt) és a uppercase pipe-ot használjuk az alábbiak szerint

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

        @Component({
          selector: 'app-birthday',
          template: `<p>Birthday is {{  birthday | date:'fullDate' | uppercase}} </p>` // THURSDAY,
JUNE 18, 1987
        })
        export class BirthdayComponent {
          birthday = new Date(1987, 6, 18);
        }
```

Mi az egyedi pipe?

A beépített pipe-on kívül mi magunk is írhatunk pipe-ot az alábbi főbb jellemzőkkel:

1.) A pipe olyan osztály, amit @Pipe decorator díszít:

```
@Pipe({name: 'myCustomPipe'})
```

2.) A pipe osztály a PipeTransform interfész transzformációs módszerét valósítja meg, amely elfogadja a bemeneti értéket, amelyet opcionális paraméterek követnek, és visszaadja az átalakított értéket. A pipeTransform szerkezete az alábbiak szerint alakul,

```
interface PipeTransform {
  transform(value: any, ...args: any[]): any
}
```

3.) A @Pipe dekorátor lehetővé teszi, hogy meghatározza a sablon kifejezésekben használt pipe nevét. Érvényes JavaScript azonosítónak kell lennie.

```
template: `{{someInputValue | myCustomPipe: someOtherValue}}`
```

Mondjon példát az egyedi pipe-ra?

```
import {Pipe, PipeTransform} from @ @ angular / core;
@Pipe({name: 'customFileSizePipe'})
export class FileSizePipe implements PipeTransform {
  transform(size: number, extension: string = 'MB'): string {
    return (size / (1024 * 1024)).toFixed(2) + extension;
  }
}
```

Template-ben a használata:

```
<h2>Find the size of a file</h2> <p>Size: {{288966 | customFileSizePipe: 'GB'}}</p>
```

Mi a különbség a pure és az impure pipe között?

A pure csövet csak akkor hívják meg, ha az Angular észleli az érték vagy a pipe-nak átadott paraméterek változását. Például egy primitív bemeneti érték (String, Number, Boolean, Symbol) vagy egy objektum hivatkozás (Date, Array, Function, Object) megváltoztatása. Az impure pipe változásdetektáláskor meghívódik függetlenül attól, hogy a paraméterek értéke változott-e vagy sem.

Mi az a bootstrapping modul?

Minden alkalmazás rendelkezik legalább egy Angular modullal, a gyökérmodult, amelyet az alkalmazás indításához kell, bootstrapping modulnak hívjuk. Közismert nevén AppModule. Az AngularCLI által létrehozott AppModule alapértelmezett szerkezete a következő lenne,

```javascript
/* JavaScript imports */
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { HttpClientModule } from '@angular/common/http';

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

/* the AppModule class with the @NgModule decorator */
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Mi az observables?

Az observables deklaratívak, amelyek támogatják az üzenetek publisher és subscriber közötti továbbítását az alkalmazásban. Főleg eseménykezelésre, aszinkron programozásra szolgálnak. Ebben az esetben egy függvény, ami az értékek közzétételéhez szolgál, de addig nem hajtódik végre, amíg a consumer fel nem iratkozik rá. A feliratkozott consumer ezután értesítéseket kap, amíg a függvény be nem fejeződik, vagy amíg le nem iratkozik.

Mi a HttpClient és milyen előnyei vannak?

A Front-end alkalmazások többsége az XMLHttpRequest felület vagy a fetch () API segítségével HTTP protokollon keresztül kommunikál a háttérszolgáltatásokkal. Az Angular egy egyszerűsített kliens HTTP API-t, HttpClient néven nyújt, amely az XMLHttpRequest felület tetején alapul. Ez az @ angular / common / http csomagból érhető el. A gyökérmodulba az alábbiak szerint importálhatjuk:

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

A HttpClient fő előnyei az alábbiak szerint sorolhatók fel:

- Tesztelhetőségi jellemzőket tartalmaz
- Típusos kérés és válasz objektumokat biztosít
- Intercept request and response
- Támogatja az Observalbe API-kat
- Támogatja a korszerű hibakezelést

Mondja el példával a HttpClient használatát?

Az alábbiakban bemutatjuk a HttpClient használatához szükséges lépéseket.

1.) A HttpClient importálása a root modulba

```
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  imports: [
    BrowserModule,
    // import HttpClientModule after BrowserModule.
    HttpClientModule,
  ],
  ......
  })
 export class AppModule {}
```

2.) Injektáljuk a HttpClient alkalmazást: Hozzunk létre példaként egy userProfileService-t (userprofile.service.ts). Adjuk meg a HttpClient get metódust is:

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

const userProfileUrl: string = 'assets/data/profile.json';

@Injectable()
export class UserProfileService {
  constructor(private http: HttpClient) { }

  getUserProfile() {
    return this.http.get(this.userProfileUrl);
  }
}
```

3.) Hozzon létre egy komponenst a service subscribe-ra: Hozzunk létre egy UserProfileComponent (userprofile.component.ts) nevű komponenst amely injektálja a UserProfileService service-t és meghívja a service metódust,

```
fetchUserProfile() {
  this.userProfileService.getUserProfile()
    .subscribe((data: User) => this.user = {
        id: data['userId'],
        name: data['firstName'],
        city:  data['city']
    });
}
```

Hogyan olvashatjuk ki a teljes választ?

Előfordulhat, hogy a válasz törzs nem adja vissza a teljes válaszadatokat, mert néha a szerverek speciális fejléceket vagy állapotkódokat is visszaadnak, amelyek fontosak az alkalmazás munkafolyamatához. A teljes válasz eléréséhez használja a HttpClient observe opciót,

```
getUserResponse(): Observable<HttpResponse<User>> {
  return this.http.get<User>(
    this.userUrl, { observe: 'response' });
}
```

Most a HttpClient.get () metódus a JSON-adatok helyett a HttpResponse-t adja vissza.

Hogyan hajtódik végre a hibakezelést?

Ha a kérés nem sikerül a szerveren, vagy hálózati problémák miatt nem éri el a szervert, akkor a HttpClient hibaobjektumot küld vissza a sikeres válasz helyett. Ebben az esetben a komponens az alábbiak szerint fog alakulni:

```
fetchUser() {
  this.userService.getProfile()
    .subscribe(
      (data: User) => this.userProfile = { ...data }, // success path
      error => this.error = error // error path
    );
}
```

Mi az RxJS?

Az RxJS egy könyvtár aszinkron és callback-alapú kód összeállításához funkcionális, reaktív stílusban az Observables segítségével. Számos API, például a HttpClient állítja elő és használja az RxJS Observable elemeket, és operátorokat is használ az observable adatok feldolgozásához.

```
import { Observable, throwError } from 'rxjs';
import { catchError, retry } from 'rxjs/operators';
```

Mi a subscribing?

Egy subscriber példány csak akkor kezdi meg az értékek közzétételét (publish), ha valaki feliratkozik (subscribe) rá. Tehát fel kell iratkoznia (subscribe) a példány Subscribe () metódusának meghívásával, egy observable objektum átadásával az értesítések fogadásához.

```
Creates an observable sequence of 5 integers, starting from 1
const source = range(1, 5);

// Create observer object
const myObserver = {
  next: x => console.log('Observer got a next value: ' + x),
  error: err => console.error('Observer got an error: ' + err),
  complete: () => console.log('Observer got a complete notification'),
};

// Execute with the observer object and Prints out each item
source.subscribe(myObserver);
// => Observer got a next value: 1
// => Observer got a next value: 2
// => Observer got a next value: 3
```

```
// => Observer got a next value: 4
// => Observer got a next value: 5
// => Observer got a complete notification
```

Mi az Observable?

Az Observable egy Promise-hoz hasonló egyedi objektum, amely segíthet az aszinkron kód kezelésében. Az Observable elemek nem részei a JavaScript nyelvének, ezért egy népszerű RxJS nevű observable könyvtárra kell támaszkodnunk.

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

const observable = new Observable(observer => {
  setTimeout(() => {
    observer.next('Hello from a Observable!');
  }, 2000);
});
```

Mi az Observer?

Az Observer egy felület az Observable által küldött push-alapú értesítések consumer-ei számára.

```
interface Observer<T> {
  closed?: boolean;
  next: (value: T) => void;
  error: (err: any) => void;
  complete: () => void;
}
```

Az kezelő, amely az Observer interfészt használja az observable értesítések fogadásához, az alábbiak paramétereként kerül továbbításra:

```
myObservable.subscribe(myObserver);
```

Mi a különbség a promise és az observable között?

Observable	Promise
Deklaratív: A kód futása csak a subscription után kezdődik, tehát akkor futtathatjuk amikor az eredményre szükség van	Végrehajtás azonnal a létrehozás után
Egyidőben több értéket is adhat	Csak egy érték
Subscribe method a hibakezelésre szolgál, amely központosított és kiszámítható hibakezelést tesz lehetővé	A gyerek promise-okra adja a hibát
Láncolást (chaining) és az összetett alkalmazáskra feliratkozást tesz lehetővé	Csak then()-t használ

Mi a multicasting?

A multi-casting a broadcasting, amikor egyetlen futással több subscriber is megkapja az üzenetet.

```
var source = Rx.Observable.from([1, 2, 3]);
var subject = new Rx.Subject();
var multicasted = source.multicast(subject);

// These are, under the hood, `subject.subscribe({...})`:
multicasted.subscribe({
  next: (v) => console.log('observerA: ' + v)
});
multicasted.subscribe({
  next: (v) => console.log('observerB: ' + v)
});

// This is, under the hood, `s
```

Hogyan végezheti el az observable hibakezelését?

A hibákat úgy kezelhetünk, hogy error callback-et határozunk meg az observer-en, ahelyett, hogy az aszinkron környezetben try / catch-re hagyatkoznánk.

```
myObservable.subscribe({
  next(num) { console.log('Next num: ' + num)},
  error(err) { console.log('Received an errror: ' + err)}
});
```

Mi az subscribe metódus?

Az Subscribe () metódus képes callback function definíciókat fogadni, mint next, error, and complete handlers.

```
myObservable.subscribe(
  x => console.log('Observer got a next value: ' + x),
  err => console.error('Observer got an error: ' + err),
  () => console.log('Observer got a complete notification')
);
```

Milyen segédfunkciókat biztosít az RxJS?

Az RxJS könyvtár az alábbi segédfunkciókat is biztosítja az observables létrehozásához és azokkal való munkához.

- Az aszinkron műveletek meglévő kódjának konvertálása observables-é
- Az értékek iterálása stream-ben
- Értékek hozzárendelése különböző típusokhoz
- Filtering streams
- Composing multiple streams

Melyek a observable creation functions?

Az RxJS creation functions biztosít a observable elemek létrehozásához, mint promises, events, timers and Ajax requests.

1.) Observable létrehozása Promise-ból:

```
import { from } from 'rxjs'; // from function
const data = from(fetch('/api/endpoint')); //Created from Promise
data.subscribe({
 next(response) { console.log(response); },
 error(err) { console.error('Error: ' + err); },
 complete() { console.log('Completed'); }
});
```

2.) Observable létrehozása, ami egy AJAX kérést hoz létre:

```
import { ajax } from 'rxjs/ajax'; // ajax function
const apiData = ajax('/api/data'); // Created from AJAX request
// Subscribe to create the request
apiData.subscribe(res => console.log(res.status, res.response));
```

3.) Számlálóból observable létrehozása:

```
import { interval } from 'rxjs'; // interval function
const secondsCounter = interval(1000); // Created from Counter value
secondsCounter.subscribe(n =>
  console.log(`Counter value: ${n}`));
```

4.) Eseményből observable létrehozása:

```
import { fromEvent } from 'rxjs';
const el = document.getElementById('custom-element');
const mouseMoves = fromEvent(el, 'mousemove');
const subscription = mouseMoves.subscribe((e: MouseEvent) => {
  console.log(`Coordnitaes of mouse pointer: ${e.clientX} * ${e.clientY}`);
  });
```

Mi történik, ha nincs handler-je az observable-nek?

Normális esetben egy observable objektum next, error and complete notification type handlers bármilyen kombinációjával definiálható. Ha nincs handler-je az observable-nek akkor figyelmen kívül hagy ilyen típusú értesítéseket.

Mik azok az Angular elemek?

Angular elemek olyan Angular komponensek, amelyek egyedi elemként vannak csomagolva (webes szabvány az új HTML elemek framework-agnostic módon történő meghatározásához). Az Angular elemek Angular komponenst tartalmaznak, amelyek hidat biztosítanak a komponensben definiált adatok és logika, valamint a szabványos DOM API-k között

Mi az Angular elemek böngészőtámogatása?

Mivel az Angular elemek egyedi elemként vannak csomagolva, az Angular elemek böngészőtámogatása megegyezik a custom elemek támogatásával.

Browser	Angular Element Support
Chrome	Natively supported
Opera	Natively supported

Browser	Angular Element Support
Safari	Natively supported
Firefox	Natively supported from 63 version onwards. You need to enable dom.webcomponents.enabled and dom.webcomponents.customelements.enabled in older browsers
Edge	Currently it is in progress

Mik az egyedi elemek (custom elements)?

Az egyedi elemek (vagy a webkomponensek) egy olyan webplatform-szolgáltatás, amely kiterjeszti a HTML-t, lehetővé téve egy olyan címke megadását, amelynek tartalmát JavaScript-kód hozza létre és vezérli. A böngésző fenntartja a meghatározott egyéni elemek CustomElementRegistry-jét, amely egy JavaScript osztályt egy HTML tag-hez társít. Jelenleg ezt a funkciót támogatja a Chrome, a Firefox, az Opera és a Safari, és más böngészőkben is elérhető polifill-eken keresztül.

Indítanom (Bootstrap) kell az egyedi elemeket?

Nem, az egyéni elemek automatikusan elindítódnak DOM-hoz való hozzáadáskor, és automatikusan megsemmisítődnek, ha eltávolítják őket a DOM-ból. Amint egy egyéni elemet hozzáadunk a DOM-hoz bármelyik oldalhoz, úgy néz ki és úgy viselkedik, mint bármely más HTML-elem.

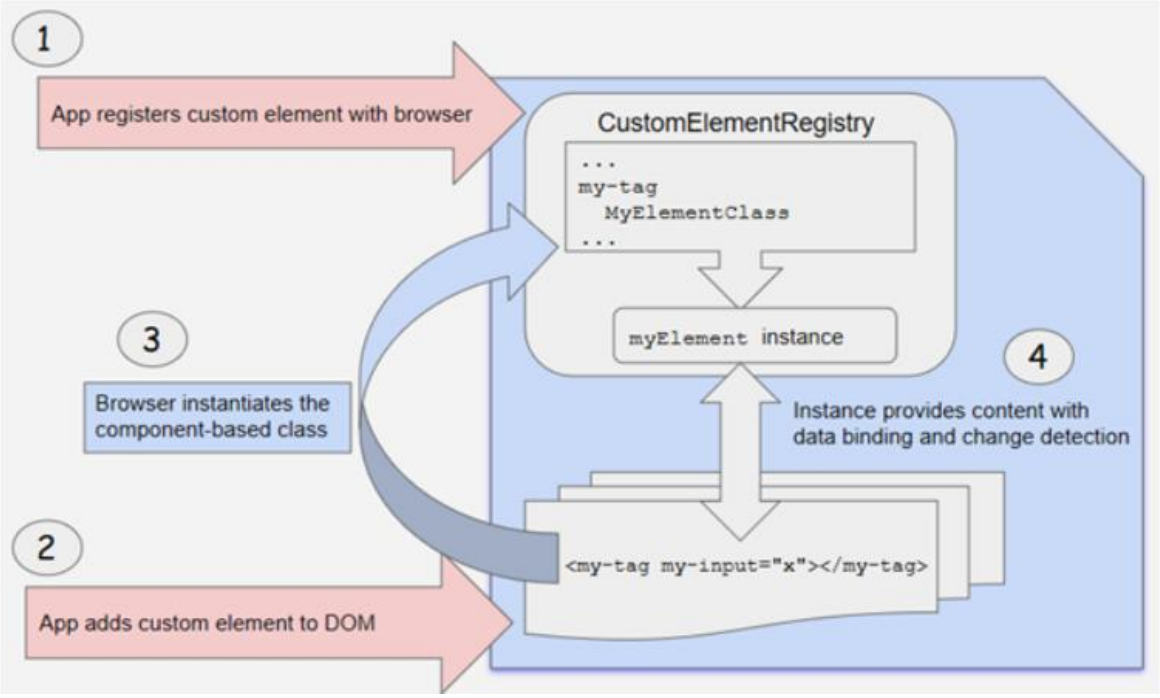
Magyarázza el, hogy az egyedi elemek hogyan működnek?

1.) Az alkalmazás regisztrálja az egyéni elemet a böngészővel: A createCustomElement () függvény segítségével konvertálhat egy komponenst olyan osztályra, amelyet egyéni elemként (custom element) regisztrálhat a böngésző.

2.) Az alkalmazás egyéni elemet (custom element) ad hozzá a DOM-hoz: Hozzáad egyéni elemet, akárcsak egy beépített HTML-elemet, közvetlenül a DOM-ba.

3.) Böngésző példányosítja az komponens alapú osztályt: A böngésző létrehozza a regisztrált osztály egy példányát, és hozzáadja a DOM-hoz.

4.) A példány adatkötéssel és változások érzékelésével látja el a tartalmat: A sablonban található tartalmat a komponens és a DOM adatok segítségével renderelik. Az egyéni elemek funkcionalitásának folyamatábrája a következő lenne

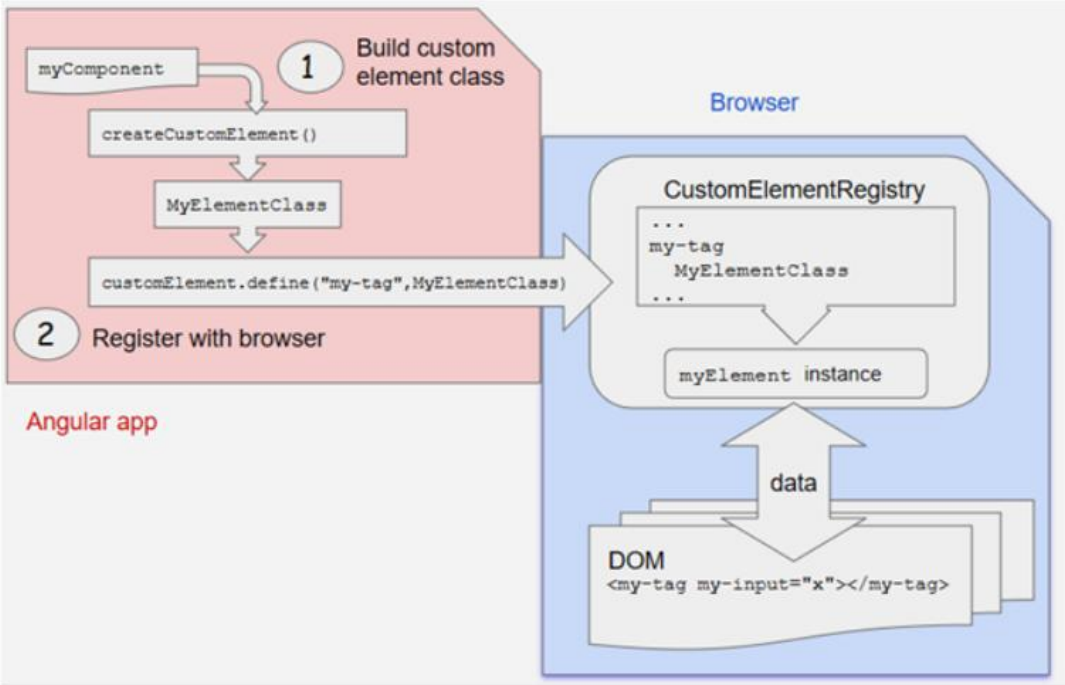


Hogyan lehet komponenseket átvinni egyéni elemekre (custom elements)?

Az komponensek átalakítása egyedi elemekké két fő lépést foglal magában,

1.) Egyéni elemosztály készítése: Az Angular a createCustomElement () függvényt biztosítja egy Angular komponens (annak függőségeivel együtt) egyéni elemmé konvertálásához. Az átalakítási folyamat megvalósítja az NgElementConstructor interfészt, és létrehoz egy konstruktor osztályt, amelyet az Angular komponens önindító példányának előállításához használnak.

2.) Elemosztály regisztrálása böngészővel: A customElements.define () JS függvényt használja a konfigurált konstruktor és a hozzá tartozó ustom-element tag regisztrálásához a böngésző CustomElementRegistry-jében. Amikor a böngésző találkozik a regisztrált elem címkéjével, akkor a konstruktort használja egy egyedi elem példány (custom-element) létrehozására.



Milyen leképezési (mapping) szabályok vannak az Angular komponens és az egyéni elem (custom tag) között?

Az Component properties és logikája közvetlenül a HTML attribútumokba és a böngésző eseményrendszerébe képez le. Az alábbi két lépésben:

1.) A createCustomElement () API elemzi az komponens bemeneti tulajdonságait az egyedi elem megfelelő attribútumaival. Például a @Input ('myInputProp') komponens, amelyet egyedi elemként alakítottunk át, a my-input-prop.

2.) A komponens kimenetek HTML egyéni eseményként kerülnek elküldésre, az egyéni esemény neve megegyezik a kimenet nevével. Például a @Output () valueChanged = new EventEmitter () komponens átalakítva egyéni elemként (custom element), a dispatch event pedig "valueChanged" néven.

typings for custom elements?

A @ angular / elements fájlból exportált NgElement és WithProperties típusokkal.

Nézzük meg, hogyan lehet alkalmazni, összehasonlítva az Angular komponenssel.

Az egyszerű konténer a input tulajdonsággal az alábbiak szerint alakul,

```
@Component(...)
class MyContainer {
  @Input() message: string;
}
```

A típusok alkalmazása után a TypeScript ellenőrzi a bemeneti értéket és típusukat,

```
const container = document.createElement('my-container') as NgElement & WithProperties<{message:
string}>;
container.message = 'Welcome to Angular elements!';
container.message = true;  // <-- ERROR: TypeScript knows this should be a string.
container.greet = 'News';  // <-- ERROR: TypeScript knows there is no `greet` property on `container`.
```

Mik a dinamikus komponensek?

A dinamikus komponensek azok az komponensek, amelyeknál az alkalmazás komponenseinek helye nincs meghatározva a készítés idején. Vagyis egyetlen Angular sablonban sem használják őket. De az komponens példányosítva lesz és futás közben kerül be az alkalmazásba.

Mik a különféle direktívák?

Háromféle direktíva létezik,

- Komponensek - Ezek sablonnal rendelkező direktívák.
- Strukturális direktívák - Ezek a direktívák megváltoztatják a DOM elrendezését a DOM elemek hozzáadásával és eltávolításával.
- Attribútum direktívák - Ezek a direktívák megváltoztatják egy elem, komponens vagy más direktíva megjelenését vagy viselkedését.

Hogyan hozhat létre direktívákat a CLI használatával?

Használhatja a CLI ng generator parancsot az osztály osztályfájl létrehozásához. Létrehozza a forrásfájlt (src / app / components / directivename.directive.ts), a megfelelő tesztfájlt (.spec.ts), és deklarálja a direktíva osztályfájlt a root modulban.

Mondjon példát az attribútum direktívákra?

Az alábbi lépésekkel hozhatja létre és alkalmazhatja az attribútum direktívát:

Hozzon létre HighlightDirective osztályt az src / app / highlight.directive.ts fájlnévvel. Ebben a fájlban importálnunk kell a direktívát a core library-ből, hogy a metaadatokat és az ElementRef-et a direktíva konstruktorában alkalmazzuk, hogy a host DOM elemre hivatkozást adjunk.

```
import { Directive, ElementRef } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
    constructor(el: ElementRef) {
       el.nativeElement.style.backgroundColor = 'red';
    }
}
```

Az attribútum direktívát alkalmazzuk a host elem attribútumaként:

```
<p appHighlight>Highlight me!</p>
```

Mi az Angular router?

Az Angular Router olyan mechanizmus, amelyben a navigáció egyik nézetről a másikra történik, miközben a felhasználók végrehajtják az alkalmazás feladatait. A böngésző alkalmazás navigációjának koncepcióját vagy modelljét veszi.

Mi a célja az base href?

Az routing alkalmazásnak hozzá kell adnia az index.html elemhez mint első gyermek, hogy jelezze, hogyan kell összeállítani a navigációs URL-eket. Ha az app mappa az alkalmazás gyökere, akkor a href értékét az alábbiak szerint állíthatja be

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

Mi az router import?

Az Angular router, amely egy adott URL-hez egy adott komponens nézetet képvisel, nem része az Angular Core-nak. A @ angular / router nevű könyvtárban elérhető a szükséges router komponensek importálásához. Például az alábbiakban importáljuk őket az app modulba,

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

Mi az a router outlet?

A RouterOutlet egy direktíva a router könyvtárból, az routernek ott kell a kimenetet megjeleníteni.

```
<router-outlet></router-outlet>
<!-- Routed components go here -->
```

Mik azok a router linkek?

A RouterLink lehetővé teszi az útválasztó (router) számára az elemek irányítását. Mivel a navigációs útvonalak rögzítettek, az alábbiakban hozzárendelhet string értékeket a router-link direktívához:

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/todosList" >List of todos</a>
  <a routerLink="/completed" >Completed todos</a>
</nav>
<router-outlet></router-outlet>
```

Mik az active router link?

A RouterLinkActive egy direktíva, amely az aktuális RouterState alapján CSS osztályokat ad hozzá, amikor a link active, és eltávolítja, ha a link inactive. Például felveheti őket a RouterLinks-be az alábbiak szerint

```
<h1>Angular Router</h1>
<nav>
  <a routerLink="/todosList" routerLinkActive="active">List of todos</a>
  <a routerLink="/completed" routerLinkActive="active">Completed todos</a>
</nav>
<router-outlet></router-outlet>
```

Mi az a router state?

A RouterState az activated routes fája. A fa minden csomópontja ismeri az "consumed" URL szegmenseket. Az aktuális RouterState-t bárhonnan elérheti az alkalmazásból a Router szolgáltatás és a routerState property használatával.

```
@Component({templateUrl:'template.html'})
class MyComponent {
  constructor(router: Router) {
    const state: RouterState = router.routerState;
    const root: ActivatedRoute = state.root;
    const child = root.firstChild;
    const id: Observable<string> = child.params.map(p => p.id);
    //...
  }
}
```

Mik az útválasztó eseményei (router events)?

Minden navigáció során a Router navigációs eseményeket (router events) ad ,a Router.events tulajdonságon keresztül lehetővé teszi az útvonal életciklusának nyomon követését. A router eseményeinek sorrendje az alábbiak szerint alakul,

i. NavigationStart,
ii. RouteConfigLoadStart,
iii. RouteConfigLoadEnd,
iv. RoutesRecognized,
v. GuardsCheckStart,
vi. ChildActivationStart,
vii. ActivationStart,
viii. GuardsCheckEnd,
ix. ResolveStart,
x. ResolveEnd,
xi. ActivationEnd
xii. ChildActivationEnd
xiii. NavigationEnd,
xiv. NavigationCancel,
xv. NavigationError
xvi. Scroll

Mi az activated root?

Az ActivatedRoute információkat tartalmaz egy útvonalról, amely egy outlet-ba van töltve. Használható az útválasztó állapotfájának áthaladására is. Az ActivatedRoute router service-ként kerül

injektálásra az információk eléréséhez. Az alábbi példában az útvonal elérési útját és paramétereit mutatjuk be,

```
@Component({...})
class MyComponent {
  constructor(route: ActivatedRoute) {
    const id: Observable<string> = route.params.pipe(map(p => p.id));
    const url: Observable<string> = route.url.pipe(map(segments => segments.join('')));
    // route.data includes both `data` and `resolve`
    const user = route.data.pipe(map(d => d.user));
  }
}
```

Hogyan készítünk útvonalakat (routes)?

Az útválasztót (routes) konfigurálni kell az útvonal-meghatározások listájával. Az útválasztót útvonallal konfigurálni a RouterModule.forRoot () metóduson keresztül kell, és hozzáadni az eredményt az AppModule importtömbjéhez.

```
const appRoutes: Routes = [
  { path: 'todo/:id',      component: TodoDetailComponent },
  {
    path: 'todos',
    component: TodosListComponent,
    data: { title: 'Todos List' }
  },
  { path: '',
    redirectTo: '/todos',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];

@NgModule({
  imports: [
    RouterModule.forRoot(
      appRoutes,
      { enableTracing: true } // <-- debugging purposes only
    )
    // other imports here
  ],
  ...
})
export class AppModule { }
```

Mi a Wildcard útvonal célja?

Ha az URL nem felel meg előre definiált útvonalaknak, akkor az útválasztót hibát dob és összeomlik az alkalmazás. Ebben az esetben Wildcard route-ot használhat. A Wildcard útvonal két csillagból áll, amelyek minden URL-hez illeszkednek.

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

Szükségem van-e mindig Routing modulra?

Nem, az Routing modul tervezési lehetőség. Kihagyhatjuk az Routing modult (például AppRoutingModule), ha a konfiguráció egyszerű, és egyesíthetjük az útválasztási konfigurációt közvetlenül a kísérő modulba (például AppModule). De akkor ajánlott, ha a konfiguráció összetett.

Mi az az Angular Universal?

Az Angular Universal egy server-side rendering module, amely @ angular / platform-server csomag alatt érhető el. A közelmúltban az Angular Universal integrálva lett az Angular CLI-vel.

Milyen különféle típusú fordítások vannak az Angularban?

Az Angularban kétféleképpen lehet az alkalmazás fordítani,

- Just-in-Time (JIT)
- Ahead-of-Time (AOT)

Mi a JIT?

A Just-in-Time (JIT) egy olyan fordítás, amely futás közben fordítja le az alkalmazást a böngészőben. A JIT-fordítás az alapértelmezett, amikor kiadjuk az az ng build (csak build) vagy ng serve (build és serve locally) CLI parancsokat. azaz a JIT fordításához használt alábbi parancsok

```
ng build
ng serve
```

Mi az AOT?

Az Ahead-of-Time (AOT) egy olyan típusú fordítás, amely build időben fordítja az alkalmazást. Az AOT fordításhoz az - aot opciót kell adni az ng build vagy ng serve parancshoz az alábbiak szerint,

```
ng build --aot
ng serve --aot
```

Az ng build parancs a –prod flag használatával (ng build --prod) alapértelmezés szerint az AOT-val fordít.

Miért van szükség fordítási folyamatra?

Az Angular komponenseket és sablonokat a böngésző nem érti közvetlenül. Ezért az Angular alkalmazásoknak fordítási folyamatra van szükségük, mielőtt futtathatnának egy böngészőben. Például az AOT-fordításban mind az Angular HTML, mind a TypeScript kód hatékony JavaScript-kóddá alakult át a build fázisban, mielőtt a böngésző futtatta volna.

Melyek az AOT előnyei?

Az alábbiakban felsoroljuk az AOT előnyeit,

1.) Gyorsabb renderelés: A böngésző letölti az alkalmazás előre lefordított verzióját. Tehát az alkalmazás lefordítása nélkül azonnal megjelenítheti az alkalmazást.

2.) Kevesebb aszinkron kérés: Külső HTML-sablonokat és CSS-stíluslapokat von be az javascript alkalmazásba, amelyek kiküszöbölik a különálló ajax kéréseket.

3.) Kisebb Angular keretrendszer letöltési mérete: Nincs szükség az Angular fordító letöltésére. Ezért drámai módon csökkenti az alkalmazás terhelését.

4.) Sablonhibák korábbi észlelése: A template binding hibákat észleli és jelentést készít maga az build során

5.) Jobb biztonság: HTML sablonokat és komponenseket állít össze JavaScript-be. Tehát nem lesznek injekciós támadások.

Milyen módszerekkel kontrollálhatjuk az AOT-build-et?

1.) A template compile beállításainak megadásával a tsconfig.json fájlban

2.) Az Angular metaadatok dekorátorokkal történő konfigurálásával

Mi az AOT három fázisa?

Az AOT fordító három fázisban működik,

1.) Kódelemzés: A fordító rögzíti a forrás reprezentációját

2.) Kódgenerálás: Kezeli az kiértékelést, valamint korlátozza a kiértékelést.

3.) Validáció: Ebben a szakaszban az Angular sablonfordító a TypeScript fordító segítségével ellenőrzi a template-ekben a binding kifejezéseket.

Használhatok arrow function-t az AOT-ban?

Nem, a arrow vagy a lambda függvényekkel nem lehet értékeket rendelni a dekorátor tulajdonságaihoz. Például a következő kódrészlet érvénytelen:

```
@Component({
  providers: [{
    provide: MyService, useFactory: () => getService()
  }]
})
```

Ennek kijavításához a következő exportált függvényként kell használni:

```
function getService(){
  return new MyService();
}

@Component({
  providers: [{
    provide: MyService, useFactory: getService
  }]
})
```

Mi a célja a metadata json fájloknak?

A metadata.json fájl a decorator metaadatainak általános szerkezetének diagramjaként kezelhető, absztrakt szintaxisfaként (abstract syntax tree - AST) ábrázolva. Az elemzési szakaszban az AOT gyűjtő beolvassa a Angular dekorátorokban rögzített metaadatokat, és metaadat-információkat ad ki .metadata.json fájlokban, egy-egy .d.ts fájlban.

Használhatok bármilyen javascript kifejezést az AOT-ban?

Nem, az AOT JavaScript funkciók egy részhalmazát érti csak. Ha egy kifejezés nem támogatott szintaxist használ, akkor a collector hibát ír a .metadata.json fájlba.

Mi a folding?

A fordító csak a metaadatokban található exportált szimbólumokra való hivatkozásokat oldhatja fel. Ahol az exportálatlan tagok egy része folding-olódik a kód létrehozása közben. A folding olyan folyamat, egy kifejezést eredményt a .metadata.json fájlba rögzíti az eredeti kifejezés helyett. Például a fordító nem hivatkozhat a selector referenciára, mert nem exportáltuk:

```
let selector = 'app-root';
@Component({
  selector: selector
})
```

Inline selector-ban lesz:

```
@Component({
      selector: 'app-root'
    })
```

Ne feledje, hogy a fordító nem tud mindent folding-olni. Például a new kulcsszóval létrehozott objektumok és függvényhívások.

Mik azok a makrók?

Az AOT fordító olyan makrókat támogat függvények vagy statikus módszerek formájában, amelyek egy kifejezést egyetlen visszatérő kifejezésben adnak vissza. Vegyünk például egy alábbi makrófüggvényt,

```
export function wrapInArray<T>(value: T): T[] {
  return [value];
}
```

Használhatjuk metaadatokon belül kifejezésként,

```
@NgModule({
  declarations: wrapInArray(TypicalComponent)
})
export class TypicalModule {}
```

Mondjon példát néhány metaadathibára?

Az alábbiakban bemutatjuk a metaadatokban előforduló hibákat,

1.) Expression form not supported: A fordító korlátozott kifejezésszintaxisán kívül eső, a Angular metaadatokban használt nyelvi funkciók egy része ezt a hibát okozhatja. Nézzük meg néhány példát

```
1. export class User { ... }
   const prop = typeof User; // typeof is not valid in metadata
2. { provide: 'token', useValue: { [prop]: 'value' } }; // bracket notation is not valid in metadata
```

2.) Hivatkozás helyi (nem exportált) szimbólumra: A fordító egy olyan helyileg definiált szimbólumra hivatkozott, amelyet vagy nem exportáltak, vagy nem inicializáltak. Vegyünk példát erre a hibára,

```
// ERROR
let username: string; // neither exported nor initialized

@Component({
  selector: 'my-component',
  template: ... ,
  providers: [
    { provide: User, useValue: username }
  ]
})
export class MyComponent {}
```

Ezt kijavíthatjuk az érték exportálásával vagy inicializálásával,

```
export let username: string; // exported
(or)
let username = 'John'; // initialized
```

3.) A függvényhívások nem támogatottak: A fordító jelenleg nem támogatja a függvény kifejezéseket vagy a lambda függvényeket. Például nem állíthatja be a provider' useFactory-jét névtelen függvényként vagy arrow function-ként

```
providers: [
    { provide: MyStrategy, useFactory: function() { ... } },
    { provide: OtherStrategy, useFactory: () => { ... } }
  ]
```

Exportált függvénnyel viszont lehet:

```
export function myStrategy() { ... }
export function otherStrategy() { ... }
... // metadata
providers: [
    { provide: MyStrategy, useFactory: myStrategy },
    { provide: OtherStrategy, useFactory: otherStrategy },
```

4.) Destructured variable or constant not supported: A fordító nem támogatja a destrukturálás által hozzárendelt változókra való hivatkozásokat. Például nem írhat ilyesmit:

```
import { user } from './user';

// destructured assignment to name and age
const {name, age} = user;
... //metadata
providers: [
    {provide: Name, useValue: name},
    {provide: Age, useValue: age},
  ]
```

De a következő érvényes:

```
import { user } from './user';
... //metadata
providers: [
    {provide: Name, useValue: user.name},
    {provide: Age, useValue: user.age},
  ]
```

Mi a metaadatok átírása (metadata rewriting)?

A metaadatok átírása az a folyamat, amelynek során a fordító a kifejezést, például a useClass, useValue, useFactory és adatokat inicializáló kifejezést egy exportált változóvá alakítja, amely helyettesíti a kifejezést. A fordító ezt az átírást a .js fájlban végzi, de a definíciós fájlokban (.d.ts fájl) nem.

Hogyan biztosítjuk a konfigurációs öröklődést (configuration inheritance)?

Az Angular Compiler támogatja a konfigurációs öröklődést (configuration inheritance) a tsconfig.json fájllal az angularCompilerOptions-ben. azaz az alapfájl konfigurációját (például tsconfig.base.json) először betöltjük, majd felülírjuk az öröklődő konfigurációs fájlban lévőkkel.

```
{
  "extends": "../tsconfig.base.json",
  "compilerOptions": {
    "experimentalDecorators": true,
    ...
  },
  "angularCompilerOptions": {
    "fullTemplateTypeCheck": true,
    "preserveWhitespaces": true,
    ...
  }
}
```

Hogyan adhatjuk meg az Angular template compiler beállításait?

Az Angular template compiler beállításait az angularCompilerOptions objektum tagjaiként adják meg a tsconfig.json fájlban.

```
{
  "compilerOptions": {
    "experimentalDecorators": true,
              ...
  },
  "angularCompilerOptions": {
    "fullTemplateTypeCheck": true,
    "preserveWhitespaces": true,
              ...
  }
}
```

Hogyan engedélyezi a binding expression validation-t?

A fullTemplateTypeCheck-re kell állítani a tsconfig.json-ban az "angularCompilerOptions"-t Hibaüzeneteket hoz létre, amikor típushibát észlelnek egy template binding kifejezésben.

```
@Component({
  selector: 'my-component',
  template: '{{user.contacts.email}}'
})
class MyComponent {
  user?: User;
}
```

Ez az alábbi hibát adja:

```
my.component.ts.MyComponent.html(1,1): : Property 'contacts' does not exist on type 'User'. Did you
mean 'contact'?
```

Mi a célja any type cast függvénynek?

A $ any () típusú cast függvény használatával letilthatjuk kifejezés ellenőrzését.

```
template:
  '{{ $any(user).contacts.email }}'
```

A $ any () cast függvény this kulcsszóval is működik, hogy a komponens nem deklarált tagjaihoz hozzáférést nyújt.

```
template:
   '{{ $any(this).contacts.email }}'
```

Mi a Non null type assertion operátor?

A Non null type assertion operátorral elnyomhatja az Object is 'undefined' hibát. A következő példában a felhasználó és az elérhetőség tulajdonságai mindig együtt vannak beállítva, ami azt jelenti, hogy a kapcsolat mindig nem null, ha a felhasználó nem null. A hibát a példában a contact! .email használatával elnyomjuk.

```
@Component({
  selector: 'my-component',
```

```
  template: '<span *ngIf="user"> {{user.name}} contacted through {{contact!.email}} </span>'
})
class MyComponent {
  user?: User;
  contact?: Contact;

  setData(user: User, contact: Contact) {
    this.user = user;
    this.contact = contact;
  }
}
```

Mi a type narrowing?

Az ngIf directivában használt kifejezést narrow type union –ként használják az Angular template-ben. Tehát a * ngIf lehetővé teszi a typeScript fordító számára, hogy arra következtessen, hogy a binding expression-ben használt adatok soha nem lesznek undefined-ek.

```
@Component({
  selector: 'my-component',
  template: '<span *ngIf="user"> {{user.contact.email}} </span>'
})
class MyComponent {
  user?: User;
}
```

Milyen Angular dependency-k vannak?

A package.json dependency része az alábbiak szerint osztható fel:

1.) Angular packages: Angular core és opcionális modulok; csomagnevük @ angular /.

2.) Support packages: Harmadik féltől származó könyvtárak, amelyeknek jelen kell lenniük az Angular alkalmazások futtatásához.

3.) Polyfill csomagok: A Polyfills kitölti a hiányosságokat a böngésző JavaScript implementációjában

Mi az a zóna (zone)?

A zóna egy végrehajtási környezet (execution context), amely az aszinkron feladatok között is fennáll. Az Angular a zone.js-re támaszkodik az Angular változásfelismerési folyamatainak futtatásához, amikor a natív JavaScript-műveletek eseményeket vetnek fel

Mi a célja a common modulnak?

A @ angular / common modul tagjai : services, pipes, and directives Ezeken kívül a HttpClientModule a @ angular / common / http –vel érhető el.

Mi a codelyzer?

A Codelyzer tslint-szabályokat tartalmaz az Angular TypeScript projektek statikus kódelemzéséhez. Futtathatjuk a statikus kódelemzőt webalkalmazásokon, NativeScript, Ionic stb. felett. Az Angular CLI támogatja ezt, és az alábbiak szerint használható

```
ng new codelyzer
ng lint
```

Mi az az Angular animáció?

Az Angular animációs rendszere a CSS funkcionalitására épül annak érdekében, hogy animáljon minden olyan tulajdonságot, amelyet a böngésző animálhatónak tart. Ezek a tulajdonságok magukban foglalják a pozíciókat, méreteket, átalakításokat, színeket, szegélyeket stb.

Mik az animációs modul használatának lépései?

Kövesse az alábbi lépéseket az animáció megvalósításához:

1.) Az animációs modul engedélyezése

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

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

2.) Animációs függvények importálása komponens fájlokba: Importálja a szükséges animációs függvényeket a @ angular / animations fájlból a komponens fájlokba (például src / app / app.component.ts).

```
import {
  trigger,
  state,
  style,
  animate,
  transition,
  // ...
} from '@angular/animations';
```

3.) Az animáció metaadat tulajdonságának hozzáadása: adjon hozzá egy animáció nevű metaadat tulajdonságot: a @Component () dekorátoron belül komponens fájlokba (például src / app / app.component.ts)

```
@Component({
  selector: 'app-root',
  templateUrl: 'app.component.html',
```

```
  styleUrls: ['app.component.css'],
  animations: [
    // animation triggers go here
  ]
})
```

Mi az a State függvény?

Az Angular sate () függvénye különböző állapotok meghatározására szolgál, amelyeket az egyes átmenetek (transition) végén hívni kell. Ennek a függvénynek két argumentuma van: egy egyedi név, például open or closed and a style() függvény.

```
state('open', style({
  height: '300px',
  opacity: 0.5,
  backgroundColor: 'blue'
})),
```

Mi a Style függvény?

A style függvény egy adott állapot névhez társítandó stíluskészlet meghatározására szolgál. A CSS stílusattribútumok beállításához az state () függvénnyel együtt kell használnia. Például close állapotban a gomb magassága 100 pixel, átlátszatlansága 0,8, a háttér színe zöld.

```
state('close', style({
  height: '100px',
  opacity: 0.8,
  backgroundColor: 'green'
})),
```

Mi az animációs fügvény célja?

Az Angular animációk hatékony módszer a kifinomult és lenyűgöző animációk megvalósítására az Angular SPA webalkalmazáshoz.

```
import { Component, OnInit, Input } from '@angular/core';
import { trigger, state, style, animate, transition } from '@angular/animations';

@Component({
selector: 'app-animate',
templateUrl: `<div [@changeState]="currentState" class="myblock mx-auto"></div>`,
styleUrls: `.myblock {
    background-color: green;
    width: 300px;
    height: 250px;
    border-radius: 5px;
    margin: 5rem;
    }`,
animations: [
    trigger('changeState', [
    state('state1', style({
        backgroundColor: 'green',
        transform: 'scale(1)'
    })),
    state('state2', style({
        backgroundColor: 'red',
        transform: 'scale(1.5)'
    })),
    transition('*=>state1', animate('300ms')),
    transition('*=>state2', animate('2000ms'))
    ])
```

```
]
})
export class AnimateComponent implements OnInit {

    @Input() currentState;

    constructor() { }

    ngOnInit() {
    }
}
```

Mi a transition függvény?

Az animációs transition függvény segítségével meghatározhatók azok a változások, amelyek egy állapot és egy másik állapot között egy bizonyos időtartam alatt bekövetkeznek. Két argumentumot fogad el: az első argumentum olyan kifejezést fogad el, amely meghatározza a két átmeneti állapot közötti irányt, a második argumentum pedig egy animate () függvény.

Vegyünk egy példát az állapotátmenetről nyitottról zártra egy fél másodperces átmenettel az állapotok között.

```
transition('open => closed', [
  animate('500ms')
]),
```

Hogyan lehet a dinamikus szkriptet Angularban injektálni?

A DomSanitizer használatával dynamic Html,Style,Script,Url-t adhatunk meg:

```
import { Component, OnInit } from '@angular/core';
import { DomSanitizer } from '@angular/platform-browser';
@Component({
   selector: 'my-app',
   template: `
       <div [innerHtml]="htmlSnippet"></div>
   `,
})
export class App {
       constructor(protected sanitizer: DomSanitizer) {}
       htmlSnippet: string = this.sanitizer.bypassSecurityTrustScript("<script>safeCode()</script>");
   }
```

Mi az a service worker és milyen szerepe van az Angularban?

A service worker s egy olyan szkript, amely a webböngészőben fut, és kezeli az alkalmazások gyorsítótárát. Az 5.0.0 verziótól kezdve az Angular tartalmazza. Az Angular Service Worker célja, hogy optimalizálja az alkalmazás lassú vagy megbízhatatlan hálózati kapcsolaton keresztül történő használatát, ugyanakkor minimalizálja az elavult tartalom kiszolgálásának kockázatát.

Melyek a service workers tervezési céljai?

1.) Gyorsítótárban tárol egy alkalmazást, akárcsak egy natív alkalmazás telepítése

2.) Egy futó alkalmazás az összes fájl ugyanazzal a verziójával fut tovább, inkompatibilis fájlok nélkül

3.) Az alkalmazás frissítésekor betölti a legújabb, teljesen gyorsítótárazott verziót

4.) A változások közzétételekor azonnal frissül a háttérben

Mi az Angular ivy?

Az Angular Ivy az Angular új render engine-je. Választhatja az Ivy előnézeti verziójának az Angular 8-as verzióját.

Engedélyezheti egy új projektben az --enable-ivy segítségével:

```
ng new ivy-demo-app --enable-ivy
```

Hozzáadhatja egy meglévő projekthez úgy, hogy hozzáadja az enableIvy beállítást a projekt tsconfig.app.json-ben angularCompilerOptions-nél.

```
{
  "compilerOptions": { ... },
  "angularCompilerOptions": {
    "enableIvy": true
  }
}
```

Milyen tulajdonságai vannak az Ivy-nek?

- Generált kód, amely futás közben könnyebben olvasható és hibakereshető
- Gyorsabb újjáépítési idő
- Jobb payload size
- Továbbfejlesztett sablontípus-ellenőrzés

Mi az Angular Language Service?

Az Angular Language Service segítségével teljesítéseket, hibákat, tippeket és navigációt kaphat az Angular sablonok belsejében, függetlenül attól, hogy külső HTML-fájlban vannak-e, vagy be vannak ágyazva annotációkba / dekorátorokba egy stringben. Képes automatikusan észlelni, hogy Angular fájlt nyit meg, elolvassa a tsconfig.json fájlt, megtalálja az alkalmazásban található összes sablont, majd az összes nyelvi szolgáltatást biztosítja.

Magyarázza el az Angular Language Service előnyeit?

1.) Automatikus kiegészítés (Autocompletion): Az automatikus kiegészítés (Autocompletion) felgyorsíthatja a fejlesztési időt azáltal, hogy kontextuális lehetőségeket és tippeket.

2.) Hibaellenőrzés: Figyelmeztethet a kód hibáira is.

3.) Navigáció: A navigáció lehetővé teszi, hogy egy komponensen, utasításon van az egér, majd kattintás és F12-vel hogy közvetlenül a definíciójához lép.

Hogyan adhat hozzá web worker-t alkalmazásához?

Az alkalmazásban bárhová teht web worker-t. Például, ha költséges számítást tartalmazó fájl az src / app / app.component.ts, akkor hozzáadhat egy web worker-t a `ng generate web-worker app` paranccsal, amely létrehozza az src / app / app.worker.ts fájlt. Ez a parancs az alábbi műveletek végrehajtására szolgál:

1.) Konfigurálja a projektet a Web Workers használatára

2.) Hozzáadja az app.worker.ts fájlt üzenetek fogadásához

```
addEventListener('message', ({ data }) => {
  const response = `worker response to ${data}`;
  postMessage(response);
});
```

3.) Az app.component.ts fájl frissítve lett a Web Workers fájljával

```
if (typeof Worker !== 'undefined') {
  // Create a new
  const worker = new Worker('./app.worker', { type: 'module' });
  worker.onmessage = ({ data }) => {
    console.log('page got message: $\{data\}');
  };
  worker.postMessage('hello');
} else {
  // Web Workers are not supported in this environment.
}
```

Milyen korlátozások vannak a web workers-el szemben?

Server-side Rendering-et használnak egyes környezetek vagy platformok (például @ angular / platform-server), és ezek nem támogatják a Web Workereket.

A Angular futtatása a web worker-rel a @ angular / platform-webworker használatával még nem támogatott az Angular CLI-ben.

Mi az az Angular CLI Builder?

Az Angular8 alkalmazásban a CLI Builder API stabil és elérhető azoknak a fejlesztőknek, akik parancsok hozzáadásával vagy módosításával kívánják testre szabni az Angular CLI-t. Például egy buildert egy teljesen új feladat végrehajtására alkalmazhat, vagy megváltoztathatja, hogy egy meglévő parancs melyik harmadik fél eszközét használja.

Mi az a builder?

Az builder függvény egy olyan funkció, amely az Architect API-t használja egy olyan összetett folyamat végrehajtására, mint például a "build" vagy a "test". A készítő kód egy npm csomagban van meghatározva. Például a BrowserBuilder futtat egy webpack build for a browser target, a KarmaBuilder pedig elindítja a Karma szervert és webpack build for unit tests.

Hogyan hívhatjuk meg a builder-t?

Az Angular CLI ng run egy adott célkonfigurációval rendelkező builder-t hív meg. A workspace konfigurációs fájlja, az angular.json, a built-in builder-ekhez alapértelmezett konfigurációit tartalmazza.

Melyek a case típusok Angular-ban?

Az Angular nagybetűk használatával megkülönbözteti a különféle típusok neveit. Az Angular az alábbiak listáját követi.

- camelCase: ymbols, properties, methods, pipe names, non-component directive selectors, constants. Például: "selectedUser"
- UpperCamelCase (vagy PascalCase): Az osztálynevek, beleértve az osztályokat, amelyek components, interfaces, NgModules, directives, and pipes határoznak meg.
- dash-case (or "kebab-case") : A fájlnevek leíró része, az component selectors kötőjeleket használnak a szavak között. Például: "app-user-list".
- UPPER_UNDERSCORE_CASE: Minden konstans. Például: "NUMBER_OF_USERS".

Melyek az osztály decorator-ok Angularban?

Az osztálydekorátor olyan dekoratőr, amely közvetlenül az osztálydefiníció előtt jelenik meg, amely az osztályt az adott típusúnak nyilvánítja, és a típusnak megfelelő metaadatokat szolgáltat

i. @Component()
ii. @Directive()
iii. @Pipe()
iv. @Injectable()
v. @NgModule()

Mik az class field decorator-ok?

Az class field decorator azok a utasítások, amelyeket közvetlenül egy property előtt deklaráltunk egy osztálydefinícióban, amely meghatározza a property típusát. Néhány példa: @input és @output,

```
@Input() myProperty;
@Output() myEvent = new EventEmitter();
```

Mik deklarálhatóak Angularban?

Deklarálható olyan osztálytípus, amelyet hozzáadhat egy NgModule deklarációs listájához. Az osztálytípusokat, például components, directives, and pipes, a modulban lehet megadni.

```
declarations: [
  YourComponent,
  YourPipe,
  YourDirective
],
```

Milyen korlátozások vonatkoznak a deklarálható osztályokra?

Az alábbi osztályokat nem deklaráljuk:

- Osztály, amelyet már egy másik NgModule deklarált
- Ngmodule classes
- Service classes
- Helper classes

Mi az a DI token?

A DI-token egy lookup token, amely a dependency provider-hez van társítva a dependency injection rendszerben. Az injektor fenntart egy internal token-provider map-et, amelyre hivatkozik, amikor függőséget kér, és a DI token a map kulcsa. Vegyünk példát a DI Token használatára,

```
const BASE_URL = new InjectionToken<string>('BaseUrl');
const injector =
   Injector.create({providers: [{provide: BASE_URL, useValue: 'http://some-domain.com'}]});
const url = injector.get(BASE_URL);
```

Mi az az Angular DSL?

A domain-specific language (DSL) egy adott alkalmazás-tartományra szakosodott számítógépes nyelv. Az Angularnak megvan a saját tartományspecifikus nyelve (DSL), amely lehetővé teszi számunkra, hogy a normál html-be Angular specifikus html-szerű szintaxist írjunk. Saját fordítója van, amely ezt a szintaxist html-be fordítja, amelyet a böngésző megért. Ezt a DSL-t az NgModules határozza meg, például animációk, űrlapok, valamint útválasztás és navigáció.

Alapvetően 3 fő szintaxis van az Angular DSL-ben.

- (): Output és DOM eseményekhez használható.
- []: Input és specifikus DOM elemattribútumokhoz használható.
- *: Structural directives (* ngFor vagy * ngIf) : megváltoztatják a DOM szerkezetét.

Mi az rxjs subject Angular-ban?

Az RxJS Subject egy különleges típusú Observable, amely lehetővé teszi hogy az Observable-ök multicasted legyenek. Míg a sima Observable unicast (minden feliratkozott Observer független futású Observable-el), a Subject-ek multicast-ok.

A Subject olyan, mint egy Observable, de multicast lehet számos Observer-nek. A Subject, mint az EventEmitters: sok listeners nyilvántartását vezetik.

```
import { Subject } from 'rxjs';

   const subject = new Subject<number>();

   subject.subscribe({
     next: (v) => console.log(`observerA: ${v}`)
   });
   subject.subscribe({
     next: (v) => console.log(`observerB: ${v}`)
   });

   subject.next(1);
   subject.next(2);
```

Mi a platform Angular-ban?

A platform az a kontextus, amelyben egy Angular alkalmazás fut. Az Angular alkalmazások leggyakoribb platformja egy webböngésző, de lehet mobileszköz operációs rendszere vagy webszerver is. A futásidejű platformot az @ angular / platform- * csomagok biztosítják, és ezek a csomagok lehetővé teszik az @ angular / core és @ angular / common alkalmazásokat különböző környezetekben végrehajtani, vagyis az Angular platformfüggetlen keretrendszerként használható különböző környezetekben, például

- A böngészőben történő futtatás közben platform-böngésző csomagot használ.
- Az SSR (kiszolgálóoldali megjelenítés) használatakor platform-kiszolgáló csomagot használ a webkiszolgáló megvalósításának biztosításához.

Hogyan válasszon ki egy elemet egy Angular sablonban?

A @ViewChild direktívával közvetlenül elérheti a view elemeit. Vegyünk egy input elemet referenciával,

```
<input #uname>
```

és definiálja a view child directive-t, és elérhetjük azt az ngAfterViewInit életciklus hook-ban:

```
@ViewChild('uname') input;

ngAfterViewInit() {
  console.log(this.input.nativeElement.value);
}
```

Hogyan lehet észlelni az route változását Angularban?

Az Angular7-ben feliratkozhat az route a változások észleléséhez.

```
this.router.events.subscribe((event: Event) => {})
```

nézzünk meg egy példát:

```
import { Component } from '@angular/core';
import { Router, Event, NavigationStart, NavigationEnd, NavigationError } from '@angular/router';

@Component({
    selector: 'app-root',
    template: `<router-outlet></router-outlet>`
})
export class AppComponent {

    constructor(private router: Router) {

        this.router.events.subscribe((event: Event) => {
            if (event instanceof NavigationStart) {
                // Show loading indicator and perform an action
            }

            if (event instanceof NavigationEnd) {
                // Hide loading indicator and perform an action
            }

            if (event instanceof NavigationError) {
                // Hide loading indicator and perform an action
                console.log(event.error); // It logs an error for debugging
            }
        });
   }
}
```

Hogyan adhatunk át a HTTP-kliensnek header-t?

Közvetlenül átadhatja az objektum-ot, vagy létrehozhatunk HttpHeaders osztályt:

```
constructor(private _http: HttpClient) {}
this._http.get('someUrl',{
   headers: {'header1':'value1','header2':'value2'}
});

(or)
let headers = new HttpHeaders().set('header1', headerValue1); // create header object
headers = headers.append('header2', headerValue2); // add a new header, creating a new object
headers = headers.append('header3', headerValue3); // add another header

let params = new HttpParams().set('param1', value1); // create params object
params = params.append('param2', value2); // add a new param, creating a new object
params = params.append('param3', value3); // add another param

return this._http.get<any[]>('someUrl', { headers: headers, params: params })
```

Mi a differential loading célja a CLI-ben?

Az Angular8 kiadástól kezdve az alkalmazásokat a CLI differential loading stratégiája alapján építik fel, és két külön csomagot építenek a telepített alkalmazás részeként.

Az első build az ES2015 szintaxist tartalmazza, amely kihasználja a modern böngészők beépített támogatásának előnyeit, kevesebb polifill-t tartalmaz, és kisebb csomagméretet eredményez.

A második build régi ES5 szintaxist tartalmaz, hogy támogassa a régebbi böngészőket az összes szükséges polyfill-el. De ez nagyobb csomagméretet eredményez.

Megjegyzés: Ezt a stratégiát több böngésző támogatására használják, de csak a böngészőnek szükséges kódot tölti be.

Az Angular támogatja a dinamikus importálást?

Igen, az Angular 8 támogatja a dinamikus importálást az router konfigurációjában. Vagyis az import utasítással később tölti be (lazy load) a modult a loadChildren módszerrel. Tévedésből adódóan, ha elírja a modul nevét, akkor is elfogadja a karakterláncot, és hibát dob a készítés ideje alatt.

```
{path: ‘user’, loadChildren: ‘./users/user.module#UserModulee’},
```

Ezt a problémát dinamikus importálás segítségével oldják meg, és az IDE-k képesek megtalálni a fordítás ideje alatt.

```
{path: ‘user’, loadChildren: () => import(‘./users/user.module’).then(m => m.UserModule)};
```

Mi a lazy loading?

A lazy loading az egyik leghasznosabb fogalma az Angular Routing. Segít abban, hogy a weboldalakat darabokban töltsük le, ahelyett, hogy mindent nagy csomagban töltenénk le. Lazy loading során aszinkron módon betöltjük a funkciómodult a routing-hoz, amikor csak szükséges a loadChildren tulajdonsággal.

```
const routes: Routes = [
  {
    path: 'customers',
    loadChildren: () => import('./customers/customers.module').then(module => module.CustomersModule)
  },
  {
    path: 'orders',
    loadChildren: () => import('./orders/orders.module').then(module => module.OrdersModule)
  },
  {
    path: '',
    redirectTo: '',
    pathMatch: 'full'
  }
];
```

Hogyan frissítheti az Angular verziót?

Az Angular frissítés meglehetősen egyszerűbb az alább említett Angular CLI ng update paranccsal.

```
ng update @angular/cli @angular/core
```

Mi az a Angular Material?

Az Angular Material az Material Design gyűjteménye a Material Design specifikációját követve.

```
npm install --save @angular/material @angular/cdk @angular/animations
(OR)
yarn add @angular/material @angular/cdk @angular/animations
```

Mi az NgUpgrade?

Az NgUpgrade az Angular csapat által összeállított könyvtár, amelyet felhasználhat az alkalmazásaiban az AngularJS és a Angular komponensek összekapcsolására dependency injection systems keresztül.

Hogyan használjuk a polifilleket Angular alkalmazásban?

Az Angular CLI hivatalosan támogatja a polyfills-eket. Amikor új projektet hoz létre az ng new paranccsal, egy src / polyfills.ts konfigurációs fájl jön létre a projektmappa részeként. Ez a fájl tartalmazza a kötelező és sok opcionális polyfills-t JavaScript importálási utasításként. Csoportosítsuk a polyfills-eket,

- Mandatory polyfills: Ezek automatikusan települnek, amikor létrehozzuk a projektet ng új paranccsal, és a megfelelő import utasításokat engedélyezzük az 'src / polyfills.ts' fájlban.
- Optional polyfills: Telepítenie kell az npm csomagot, majd import-okat kell létrehoznia az 'src / polyfills.ts' fájlban. Például először az npm alatti csomagot kell telepítenie a webanimációk (opcionális) polyfill hozzáadásához. npm install --save web-animations-js létrehozása és importálási utasítás polyfill fájlban: javascript import 'web-animations-js';

Milyen módon lehet trigger change detection (változás detektálást) in Angular?

Injektálhatjuk az ApplicationRef vagy az NgZone, vagy a ChangeDetectorRef elemeket a komponensébe, azaz 3 lehetséges mód van,

- ApplicationRef.tick (): Hívja meg ezt a metódust a változás észlelésének feldolgozására. Ellenőrzi a teljes komponensfát.
- NgZone.run (callback): Kiértékeli a callback függvényt.
- ChangeDetectorRef.detectChanges (): Csak a komponenseket és a gyerekeket érzékeli.

What are the differences of various versions of Angular?

There are different versions of Angular framework. Let's see the features of all the various versions,

i. **Angular 1:**
   - Angular 1 (AngularJS) is the first angular framework released in the year 2010.
   - AngularJS is not built for mobile devices.
   - It is based on controllers with MVC architecture.

ii. **Angular 2:**
   - Angular 2 was released in the year 2016. Angular 2 is a complete rewrite of Angular1 version.

- The performance issues that Angular 1 version had has been addressed in Angular 2 version.
- Angular 2 is built from scratch for mobile devices unlike Angular 1 version.
- Angular 2 is components based.

iii. **Angular 3:**
- The following are the different package versions in Angular 2:
  - @angular/core v2.3.0
  - @angular/compiler v2.3.0
  - @angular/http v2.3.0
  - @angular/router v3.3.0
- The router package is already versioned 3 so to avoid confusion switched to Angular 4 version and skipped 3 version.

iv. **Angular 4:**
- The compiler generated code file size in AOT mode is very much reduced.
- With Angular 4 the production bundles size is reduced by hundreds of KB's.
- Animation features are removed from angular/core and formed as a separate package.
- Supports Typescript 2.1 and 2.2.
- Angular Universal
- New HttpClient

v. **Angular 5:**
- Angular 5 makes angular faster. It improved the loading time and execution time.
- Shipped with new build optimizer.
- Supports Typescript 2.5.
- Service Worker

vi. **Angular 6:**
- It is released in May 2018.
- Includes Angular Command Line Interface (CLI), Component Development KIT (CDK), Angular Material Package, Angular Elements.
- Service Worker bug fixes.
- i18n
- Experimental mode for Ivy.
- RxJS 6.0
- Tree Shaking

vii. **Angular 7:**
- It is released in October 2018.
- TypeScript 3.1
- RxJS 6.3
- New Angular CLI
- CLI Prompts capability provide an ability to ask questions to the user before they run. It is like interactive dialog between the user and the CLI
- With the improved CLI Prompts capability, it helps developers to make the decision. New ng commands ask users for routing and CSS styles types(SCSS) and ng add @angular/material asks for themes and gestures or animations.

viii. **Angular 8:**
- It is released in May 2019.
- TypeScript 3.4

ix. **Angular 9:**

- It is released in February 2020.
- TypeScript 3.7
- Ivy enabled by default

x. **Angular 10:**
- It is released in June 2020.
- TypeScript 3.9
- TSlib 2.0

Melyek a biztonsági elvek Angular-ban?

- Kerülje a DOM API-k közvetlen használatát.
- Engedélyezze a Content Security Policy (CSP), és konfigurálja a webkiszolgálót a megfelelő CSP HTTP fejlécek visszaadására.
- Használja az offline sablon fordítót.
- Használjon Server Side XSS Protection-t.
- Használja a DOM Sanitizer-t.
- Meg kell akadályoznia a CSRF vagy XSRF támadásokat.

Hogyan kérdezzük le a CLI verziót?

```
ng v
ng version
ng -v
ng --version
```

Melyek az Angular biztonsággal kapcsolatos legjobb gyakorlatok?

- Használja a legfrissebb Angular library kiadásokat
- Kerülje a dokumentációban „Biztonsági kockázatként” megjelölt API-kat.

Mi az Angular biztonsági modell az XSS támadások megelőzéséhez?

Az Angular alapértelmezés szerint az összes értéket nem megbízhatóként kezeli, vagyis meghiúsítja a nem megbízható értékeket, ha egy értéket egy sablonból illesztenek be a DOM-ba tulajdonság, attribútum, stílus, osztálykötés vagy interpoláció útján.

Mi a template compiler szerepe az XSS támadások megelőzésében?

Az offline sablonfordító(template compiler) megakadályozza a sabloninjekció (template injection) okozta sebezhetőségeket, és jelentősen javítja az alkalmazások teljesítményét. Ezért ajánlott offline sablonfordítót használni a production deployment-nél anélkül, hogy dinamikusan generálna sablonokat.

Melyek a security contexts in Angular?

- HTML: Akkor használatos, ha egy értéket HTML-ként értelmezünk, például a innerHtml kötünk.
- Style: Akkor alkalmazzák, amikor a CSS-t a style property-ba kapcsoljuk.
- URL: Olyan URL-tulajdonságokhoz használják, mint például a <a href>.
- Resource URL: Ez egy URL, amelyet betöltenek és végrehajtanak kódként, például <script src>.

Mi az innerHTML célja?

A innerHtml a HTML-Elements property-je, amely lehetővé teszi a html-tartalom beállítását.

```
<div [innerHTML]="htmlSnippet"></div>


export class myComponent {
  htmlSnippet: string = '<b>Hello World</b>, Angular';
}
```

Sajnos ez helytelen használat esetén a Cross Site Scripting (XSS) biztonsági hibákat okozhat.

Mi a különbség az interpolated content and innerHTML között?

Az interpolated és innerHTML közötti fő különbség a kód értelmezése. Interpolated tartalom esetén a HTML-t nem értelmezi a böngésző, és a böngésző szögletes zárójeleket jelenít meg az elem szöveges tartalmában. innerHTML binding során a tartalmat értelmezzük, vagyis a böngésző <és> karaktereket HTMLEntity-ként alakít át. Például a sablonban az alábbiak szerint lehet használni,

```
<p>Interpolated value:</p>
<div >{{htmlSnippet}}</div>
<p>Binding of innerHTML:</p>
<div [innerHTML]="htmlSnippet"></div>
```

és a property a komponensben:

```
export class InnerHtmlBindingComponent {
  htmlSnippet = 'Template <script>alert("XSS Attack")</script> <b>Code attached</b>';
}
```

Annak ellenére, hogy a innerHTML binding esélyt teremt az XSS támadásra, az Angular nem biztonságosnak ismeri.

Hogyan akadályozhatja meg az automatic sanitization?

Néha az alkalmazásoknak futtatható kódot kell tartalmazniuk, például az <iframe> URL-ből való megjelenítését. Ebben az esetben meg kell akadályoznia az automatic sanitization-t az Angularban azzal, hogy azt ellenőrizzük. Alapvetően 2 lépést tartalmaz,

1.) Injektálja a DomSanitizer-t: A DomSanitizer-t a komponensbe injektálhatjuk a konstruktor paramétereként

2.) Jelöljük meg a megbízható értéket az alábbiak egyikével:

- bypassSecurityTrustHtml
- bypassSecurityTrustScript
- bypassSecurityTrustStyle
- bypassSecurityTrustUrl
- bypassSecurityTrustResourceUrl

Például az URL használata a megbízható URL-ként az alábbiak szerint alakul,

```
constructor(private sanitizer: DomSanitizer) {
  this.dangerousUrl = 'javascript:alert("XSS attack")';
  this.trustedUrl = sanitizer.bypassSecurityTrustUrl(this.dangerousUrl);
```

Biztonságos-e a direct DOM API használata?

Nem, a beépített böngésző DOM API-k nem védenek meg automatikusan a biztonsági résekkel szemben. Ebben az esetben a DOM-mal való közvetlen interakció helyett Angular template használata ajánlott. Ha ez elkerülhetetlen, akkor használja a beépített Angular sanitization függvényeket használjuk.

Mi a DOM sanitizer?

A DomSanitizer arra szolgál, hogy megelőzze a Cross Site Scripting biztonsági hibákat (XSS) azáltal, hogy az értékeket megtisztítja, hogy a különböző DOM-es környezetben biztonságosan használhatók legyenek.

Hogyan támogatja a szerveroldali XSS védelmet az Angular?

A server-side XSS -védelmet egy Angular alkalmazás támogatja egy olyan sablonnyelv használatával, amely automatikusan elkerüli az értékeket, hogy megakadályozza a kiszolgáló XSS-biztonsági réseit. De ne használjon sablonnyelvet az Angular sablonok létrehozásához a szerver oldalon, mert nagy kockázatot jelent a template-injection-nel kapcsolatosan.

Az Angular megakadályozza a http-szintű sebezhetőségeket?

Az Angular beépített támogatással rendelkezik a http szintű sebezhetőségek megelőzésére, mint például a cross-site request forgery (CSRF vagy XSRF) és a cross-site script inclusion (XSSI). Annak ellenére, hogy ezeket a sérülékenységeket szerveroldalon kell enyhíteni, az Angular segítõket nyújt az integráció megkönnyítésére a kliens oldalon.

1.) A HttpClient támogatja az XSRF támadások megakadályozására használt token mechanizmust

2.) A HttpClient könyvtár felismeri az JSON válaszok konvencióját (amely nem futtatható js kód ")]}',\n" karakterekkel, és automatikusan törli a ")]}',\n" karakterláncot az összes válaszról.

Mik azok a Http interceptorok?

A Http interceptorok a @ angular / common / http részét képezik, amelyek ellenőrzik és átalakítják az HTTP-kérelmeket az alkalmazásból a szerverre és fordítva a HTTP-válaszok során. Ezek az interceptor-ok különféle implicit feladatokat hajthatnak végre, a hitelesítéstől a naplózásig.

A HttpInterceptor interfész szintaxisa az alábbiak szerint néz ki,

```
interface HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>>
}
```

interceptor-okat használhat egy olyan service osztály deklarálásával, amely megvalósítja a HttpInterceptor interface intercept() metódusát.

```
@Injectable()
export class MyInterceptor implements HttpInterceptor {
    constructor() {}
    intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
        ...
    }
}
```

Majd használhatjuk a modul-ban:

```
@NgModule({
    ...
    providers: [
        {
            provide: HTTP_INTERCEPTORS,
            useClass: MyInterceptor,
            multi: true
        }
    ]
    ...
})
export class AppModule {}
```

Melyek a HTTP interceptor-ok alkalmazásai?

- Authentication
- Logging
- Caching

- Fake backend
- URL transformation
- Modifying headers

Multiple interceptor támogatott-e Angularban?

Igen, az Angular egyszerre több interceptor-t is támogat.

```
providers: [
  { provide: HTTP_INTERCEPTORS, useClass: MyFirstInterceptor, multi: true },
  { provide: HTTP_INTERCEPTORS, useClass: MySecondInterceptor, multi: true }
],
```

Az interceptor-okat abban a sorrendben hívják meg, amelyben rendelkezésre bocsátották őket. azaz a MyFirstInterceptort hívják meg először a fenti konfigurációban.

Hogyan használhatom az interceptor-t egy teljes alkalmazásban?

Használhatja ugyanazt a HttpInterceptors példányt az egész alkalmazásban, ha a HttpClientModule-t az AppModule-ba importálja, és az e interceptor-okat hozzáadjuk a root application injector. Például definiálunk egy osztályt, amely injektálható a root alkalmazásban.

```
@Injectable()
export class MyInterceptor implements HttpInterceptor {
  intercept(
    req: HttpRequest<any>,
    next: HttpHandler
  ): Observable<HttpEvent<any>> {

    return next.handle(req).do(event => {
      if (eventt instanceof HttpResponse) {
           // Code goes here
      }
    });

  }
}
```

Ezt követően importálja a HttpClientModule-t az AppModule-ba

```
@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, HttpClientModule],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: MyInterceptor, multi: true }
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Hogyan egyszerűsíti az Angular a nemzetköziesítést?

- Dátumok, szám, százalékok és pénznemek megjelenítése helyi formátumban.

- Szöveg előkészítése a komponens sablonokban fordításra.
- Többes számú szavak kezelése.
- Alternatív szöveg kezelése.

Hogyan regisztráljuk manuálisan a locale adatokat?

Alapértelmezés szerint az Angular csak az en-USA területi adatait tartalmazza. De ha másik területi beállítást szeretne beállítani, akkor importálnia kell az új területi beállításokat. Ezt követően regisztrálhat a registerLocaleData metódussal:

```
registerLocaleData(data: any, localeId?: any, extraData?: any): void
```

Például importáljuk a német nyelvet és regisztráljuk az alkalmazásban

```
import { registerLocaleData } from '@angular/common';
import localeDe from '@angular/common/locales/de';

registerLocaleData(localeDe, 'de');
```

Mi a sablonfordítás négy fázisa?

Az i18n sablonfordítási folyamatnak négy fázisa van:

1.) Statikus szöveges üzenetek megjelölése fordításhoz a komponens sablonokban: Az i18n minden elem tag-ra elhelyezhető, amelynek szövegét le kell fordítani. Például

```
<h1 i18n>Hello i18n!</h1>
```

2.) Fordítási fájl létrehozása: Az Angular CLI xi18n paranccsal

```
ng xi18n
```

A fenti parancs létrehoz egy messages.xlf nevű fájlt a projekt gyökérkönyvtárában.

Megjegyzés: A kibontott fájl formátumának, nevének, helyének megváltoztatásához parancsbeállításokat adhat meg.

3.) A létrehozott fordítási fájl szerkesztése: Fordítsa le a szöveget a célnyelvre. Ebben a lépésben hozzon létre egy lokalizációs mappát (például locale néven) a gyökérkönyvtárban (src), majd hozzon létre célnyelvi fordítási fájlt az alapértelmezett messages.xlf fájl másolásával és átnevezésével. Például a fordítási fájlt (messages.de.xlf) német nyelvre.

```
<trans-unit id="greetingHeader" datatype="html">
  <source>Hello i18n!</source>
  <target>Hallo i18n !</target>
  <note priority="1" from="description">A welcome header for this sample</note>
  <note priority="1" from="meaning">welcome message</note>
</trans-unit>
```

4.) A fordítási fájl egyesítése az alkalmazásba: Az Angular CLI build parancsot kell használnia az alkalmazás fordításához, egy területi beállítások kiválasztásával vagy a következő parancsopciók megadásával.

--i18nFile = elérési út a fordítási fájlhoz

--i18nFormat = a fordítási fájl formátuma

--i18nLocale = területi azonosító

Mi az i18n attribútum célja?

Az Angular i18n attribútum a lefordítható tartalmat jelöli. Ez egy egyéni attribútum, amelyet az Angular fordítók felismernek. A fordító fordítás után eltávolítja.

Megjegyzés: Ne feledje, hogy az i18n nem direktíva.

Mi a célja az egyedi azonosítónak (custom id)?

Amikor megváltoztatja a lefordítható szöveget, az Angular extractor eszköz új azonosítót generál az adott fordítási egység számára. Ezen viselkedés miatt ezt követően minden alkalommal frissítenie kell a fordítási fájlt az új azonosítóval.

Például az messages.de.xlf.html fordítófájl fordítási-egységet (trans-unit) generált néhány szöveges üzenethez, az alábbiak szerint

```
<trans-unit id="827wwe104d3d69bf669f823jjde888" datatype="html">
```

Az id attribútum manuális frissítését elkerülheti, ha egyéni azonosítót ad meg az i18n attribútumban a @@ előtag használatával.

```
<h1 i18n="@@welcomeHeader">Hello i18n!</h1>
```

Mi történik, ha az azonosító nem egyedi?

Az azonosítóknak egyedinek kell lennie. Ha ugyanazt az azonosítót használja két különböző szöveges üzenethez, akkor csak az elsőt veszi figyelembe.

Például definiáljuk ugyanazt az egyedi azonosítót myCustomId két üzenethez,

```
<h2 i18n="@@myCustomId">Good morning</h3>
<!-- ... -->
<h2 i18n="@@myCustomId">Good night</p>
```

és az első szöveg számára generált fordítási egység német nyelvre

```
<trans-unit id="myId" datatype="html">
  <source>Good morning</source>
  <target state="new">Guten Morgen</target>
</trans-unit>
```

Mivel az egyedi azonosító megegyezik, a fordítás mindkét eleme ugyanazt a szöveget tartalmazza, mint az alábbiakban

```
<h2>Guten Morgen</h2>
<h2>Guten Morgen</h2>
```

Lefordíthatok szöveget elem létrehozása nélkül?

Igen, elérheti a <ng-container> attribútum használatával. Általában a fordításhoz be kell csomagolnia egy szöveges tartalmat i18n attribútummal. De ha nem csak a fordítás kedvéért szeretne új DOM elemet létrehozni, akkor a szöveget egy elembe csomagolhatja.

```
<ng-container i18n>I'm not using any DOM element for translation</ng-container>
```

Ne feledje, hogy a <ng-container> átalakul HTML kommentté.

Hogyan tudom lefordítani az attribútumot?

Az attribútumokat lefordíthatja az i18n-x attribútum csatolásával, ahol x a lefordítandó attribútum neve. Például lefordíthatja az img src attribútumot az alábbiak szerint,

```
<img [src]="example" i18n-title title="Internationlization" />
```

Hogyan adhatunk meg a build-konfigurációt több nyelvűség számára?

Az Angular.json fájl konfigurációs beállításaiban megadhatja az összeállítási konfigurációt, például a fordítási fájl elérési útját, nevét, formátumát és az alkalmazás URL-jét. Például az alkalmazás német verziója a következőképpen:

```
"configurations": {
  "de": {
    "aot": true,
    "outputPath": "dist/my-project-de/",
    "baseHref": "/fr/",
    "i18nFile": "src/locale/messages.de.xlf",
    "i18nFormat": "xlf",
    "i18nLocale": "de",
    "i18nMissingTranslation": "error",
  }
```

Hogyan választunk ki egy elemet a komponens template-ben?

Bármely DOM elemet vezérelhet az ElementRef-en keresztül úgy, hogy injektálja azt a komponens konstruktorába. vagyis a komponensnek:

```
constructor(myElement: ElementRef) {
```

```
   el.nativeElement.style.backgroundColor = 'yellow';
}
```

Hogyan használja a jquery-t az Angular-ban?

1.) Telepítse a dependency-t: Először telepítse a jquery dependency-t az npm használatával

```
npm install --save jquery
```

2.) Adja hozzá a jquery script-et: Az Angular-CLI projektben adja hozzá a jquery relatív elérési útját az angular.json fájlban

```
"scripts": [
   "node_modules/jquery/dist/jquery.min.js"
]
```

3.) Használja a jQuery-t:

```
<div id="elementId">
  <h1>JQuery integration</h1>
</div>
import {Component, OnInit} from '@angular/core';

declare var $: any; // (or) import * as $ from 'jquery';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  ngOnInit(): void {
    $(document).ready(() => {
      $('#elementId').css({'text-color': 'blue', 'font-size': '150%'});
    });
  }
}
```

Mi az a RouteState?

A RouteState egy olyan interfész, amely a router állapotát az activated routes fájaként ábrázolja.

```
interface RouterState extends Tree {
  snapshot: RouterStateSnapshot
  toString(): string
}
```

Az aktuális RouterState-t bárhonnan elérheti az Angular alkalmazásból a Router service és a routerState tulajdonság használatával.

Mi a hidden property célja?

A hidden property a DOM elem megjelenítésére vagy elrejtésére szolgál, egy kifejezés alapján.

```
<div [hidden]="!user.name">
```

```
  My name is: {{user.name}}
</div>
```

Mi a különbség az ngIf és a hidden property között?

A fő különbség az, hogy a * ngIf eltávolítja az elemet a DOM-ból, míg a [hidden] valójában a display: none beállítást használja (a CSS-stílussal). Általában költséges a hozzáadása és eltávolítása a DOM-ból.

Mi a slice pipe?

A slice pipe új tömböt vagy karakterláncot hoz létre, amely az elemek egy részhalmazát (szeletét) tartalmazza. A szintaxis az alábbiak szerint néz ki,

```
{{ value_expression | slice : start [ : end ] }}
```

Például a hello kiszedése a listából:

```
@Component({
  selector: 'list-pipe',
  template: `<ul>
    <li *ngFor="let i of greeting | slice:0:5">{{i}}</li>
  </ul>`
})
export class PipeListComponent {
  greeting: string[] = ['h', 'e', 'l', 'l', 'o', 'm','o', 'r', 'n', 'i', 'n', 'g'];
}
```

Mi az index property az ngFor direktívában?

Az NgFor direktíva index property-jét használjuk az elem indexének visszaadására az egyes iterációkban.

```
<div *ngFor="let todo of todos; let i=index">{{i + 1}} - {{todo.name}}</div>
```

Mi az ngFor trackBy célja?

Az * ngFor a trackBy opcióval történő használatának fő célja a teljesítmény optimalizálása. Normál esetben, ha az NgFor-ot nagy adathalmazokkal történik, akkor egy elem eltávolítása vagy hozzáadása kiválthat nagyszámú DOM-manipulációkat. Ebben az esetben az Angular csak egy új listát lát az új objektum hivatkozásokról, és a régi DOM elemeket új DOM elemre cseréli. Segíthet az Angularnak a hozzáadott vagy eltávolított elemek nyomon követésében egy trackBy függvény biztosításával, amely az indexet és az aktuális elemet argumentumként veszi fel, és vissza kell adnia az elem egyedi azonosítóját.

```
<div *ngFor="let todo of todos; trackBy: trackByTodos">
  ({{todo.id}}) {{todo.name}}
</div>
```

a trackByTodos metódust definiálva:

```
trackByTodos(index: number, item: Todo): number { return todo.id; }
```

Mi a célja az ngSwitch direktívának?

Az NgSwitch direktíva hasonló a JavaScript switch utasításhoz, amely egy feltétel alapján egy elemet jelenít meg a lehetséges elemek közül. Ebben az esetben csak a kiválasztott elem kerül a DOM-ba. Az NgSwitch, NgSwitchCase és NgSwitchDefault direktívákkal együtt használták.

```
<div [ngSwitch]="currentBrowser.name">
  <chrome-browser    *ngSwitchCase="'chrome'"    [item]="currentBrowser"></chrome-browser>
  <firefox-browser   *ngSwitchCase="'firefox'"     [item]="currentBrowser"></firefox-browser>
  <opera-browser     *ngSwitchCase="'opera'"  [item]="currentBrowser"></opera-browser>
  <safari-browser     *ngSwitchCase="'safari'"   [item]="currentBrowser"></safari-browser>
  <ie-browser  *ngSwitchDefault           [item]="currentItem"></ie-browser>
</div>
```

Lehetséges az aliasing az input és outpur-ra?

Igen, a bemenetek és kimenetek elnevezését kétféleképpen lehet elvégezni.

Aliasing in metadata: A metaadatok be- és kimenetei kettősponttal elválasztott (:) karakterláncot használva, bal oldalon a direktívanévvel, a jobb oldalon pedig a nyilvános alias-sal. azaz a propertyName: alias formátumban:

```
inputs: ['input1: buyItem'],
outputs: ['outputEvent1: completedEvent']
```

Aliasing @Input () / @ Output () decoratorral

```
@Input('buyItem') input1: string;
@Output('completedEvent') outputEvent1 = new EventEmitter<string>();
```

Mi a safe navigation operator?

A safe navigation operator (?) (Vagy más néven Elvis Operator) arra szolgál, hogy megvédjen a null és undefined értékektől, azaz visszaadja az objektum értékét, ha létezik, különben a null értéket adja vissza.

```
<p>The user firstName is: {{user?.fullName.firstName}}</p>
```

Ezen biztonságos navigációs operátor használatával az Angular framework leállítja a kifejezés kiértékelését, amikor az eléri az első null értéket, és hiba nélkül rendereli a nézetet.

Mi a template expression -operátorok listája?

- Pipe operator
- Safe navigation operator
- Non-null assertion operator

Mi az a bootstrapped komponens?

Egy olyan belépési komponens, amelyet az Angular betölt a DOM-ba a rendszerindítási folyamat vagy az alkalmazás indítási ideje alatt. Általában ezt bootstrapped vagy root component AppComponent néven nevezik el a gyökérmodulban az alábbiak szerint.

```
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule,
    HttpClientModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent] // bootstrapped entry component need to be declared here
})
```

Hogyan indíthat (bootstrap) manuálisan egy alkalmazást?

Használhatja az ngDoBootstrap hook-ot az alkalmazás manuális indításához, ahelyett, hogy a @NgModule annotációban használnánk a bootstrap tömböt. Ez a hook a DoBootstap felület része.

```
interface DoBootstrap {
  ngDoBootstrap(appRef: ApplicationRef): void
}
```

A modult a fenti interfésznek kell megvalósítania, hogy a hook-ot a rendszerindításhoz használhassa.

```
class AppModule implements DoBootstrap {
  ngDoBootstrap(appRef: ApplicationRef) {
    appRef.bootstrap(AppComponent); // bootstrapped entry component need to be passed
  }
}
```

Szükséges, hogy a bootstrapped component entry component legyen?

Igen. A rendszerindítási folyamat ugyanis elengedhetetlen folyamat.

Mi a routed entry component?

Az útválasztó konfigurációjában hivatkozott komponenseket routed entry components-nek hívják. Ez az útválasztási bejegyzés összetevője az alábbiakban meghatározott útvonal-meghatározásban került meghatározásra:

```
const routes: Routes = [
```

```
  {
    path: '',
    component: TodoListComponent // router entry component
  }
];
```

Mivel az router definíció megköveteli, hogy a komponenst két helyhez is hozzáadjuk (router és entryComponents), ezek a komponensek mindig entry komponensek.

Megjegyzés: A fordítók elég okosak ahhoz, hogy felismerjék a router definícióját, és automatikusan hozzáadják a router komponenseket az entryComponents részhez.

Az összes komponens a production build során lebuild-elődik?

Nem, csak a belépési (entry) és a template komponensek jelennek meg a production buildekben. Ha egy komponens nem entry komponens, és nem található meg egy template-ben sem, akkor eldobásra kerül. Ezért a csomag (bundle) méretének csökkentése érdekében feltétlenül csak azok a komponensek legyenek belépési (entry) komponenseket amelyek ténylegesen is azok.

Mi az Angular fordító?

Az Angular fordítót az alkalmazás kódjának JavaScript kóddá alakítására használják. Értelmezi a sablon jelölését, egyesíti azt a megfelelő komponens osztály kóddal, és komponens factory-kat bocsát ki, amelyek létrehozzák a komponens JavaScript reprezentációját a @ Component metaadatok elemeivel együtt.

Mi a szerepe az ngModule metaadatoknak a fordítási folyamatban?

A @NgModule metaadatok arra szolgálnak, hogy elmondják az Angular fordítónak, hogy milyen komponenseket kell fordítani ehhez a modulhoz, és hogyan kell összekapcsolni ezt a modult más modulokkal.

Hogyan találja meg az Angular a components, directives és pipes-okat?

Angular fordító akkor talál komponenst vagy utasítást a sablonban, ha egyezik a selector-ja az adott komponensnek vagy direktívának. Míg egy pipe-ot talál, ha a pipe neve megjelenik a HTML sablon pipe szintaxisában.

Mondjon néhány példát az NgModules-ra?

Az Angular core könyvtárak és harmadik féltől származó könyvtárak NgModules formátumban érhetők el.

Az olyan Angular könyvtárak, mint a FormsModule, a HttpClientModule és a RouterModule, NgModules.

Számos harmadik féltől származó könyvtár, például a Material Design, az Ionic és az AngularFire2 NgModules.

Mik azok a feature module-ok?

A feature module-ok NgModules, amelyeket a kód szervezésére használnak. A feature module az Angular CLI segítségével hozható létre a gyökérkönyvtárban található alábbi paranccsal,

```
ng generate module MyCustomFeature //
```

Az Angular CLI létrehoz egy my-custom-feature nevű mappát, benne a my-custom-feature.module.ts nevű fájllal, a következő tartalommal

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';

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

Melyek az importált modulok a CLI által létrehozott feature modulokban?

A CLI által létrehozott feature modulban két JavaScript importálási utasítás található a fájl tetején

- NgModule: Az InOrder a @NgModule decorator használatához
- CommonModule: Számos általános direktívát tartalmaz, mint például az ngIf és az ngFor.

Mi a különbség az ngmodule és a javascript modul között?

NgModule	JavaScript module
Az NgModule csak a deklarálható osztályokat korlátozza	Nincsenek korlátok az osztályokra
a modul osztályait csak a deklarációk tömbjében sorolja fel	Meghatározhatja az összes tagosztályt egy óriási fájlban
Csak a hozzá tartozó deklarálható osztályokat exportálja, vagy más modulokból importálja	Bármely osztályt exportálhat
Bővíti a teljes alkalmazást service-ekkel, hozzáadva a services a tömbhöz	Az alkalmazást nem lehet szolgáltatásokkal (service) bővíteni

Melyek a deklarációk lehetséges hibái?

- Ha egy komponenst deklarálás nélkül használ, az Angular hibaüzenetet ad vissza.
- Ha megpróbálja deklarálni ugyanazt az osztályt egynél több modulban, akkor a fordító hibát ad ki.

Melyek a deklarációs elemek használatának lépései?

1. Hozzuk létre az elemet (komponens,directive, pipe), és exportáljuk abból a fájlból, ahová írtuk
2. Importáljuk a megfelelő modulba.
3. Deklaráljuk a @NgModule deklarációk tömbben.

Melyek a feature modulok?

1.) Domain: Felhasználói élmény biztosítása egy adott alkalmazás tartományhoz (például megrendelés leadása, regisztráció stb.)

2.) Routed: Ezek olyan domain jellemző modulok, amelyek legfőbb összetevői az útválasztó navigációs útvonalainak (router navigation routes) célpontjai.

3.) Routing: routing konfigurációt biztosít egy másik modul számára.

4.) Szolgáltatás (Service): Segédprogramokat (utility services) nyújt, például adatelérést (data access) és üzenetküldést (messaging) (például HttpClientModule)

5.) Widget: Komponenseket, direktívákat és pipe-okat tesz elérhetővé a külső modulok számára (például harmadik féltől származó könyvtárak, például Material UI)

Mi a provider?

A provider egy utasítás a Dependency Injection rendszernek arra vonatkozóan, hogyan lehet értéket szerezni egy függőséghez (más néven létrehozott szolgáltatások). A szolgáltatás az alábbi Angular CLI segítségével:

```
ng generate service my-service
```

A CLI által létrehozott szolgáltatás az alábbiak szerint alakul,

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

@Injectable({
  providedIn: 'root', //Angular provide the service in root injector
})
export class MyService {
}
```

Mi az ajánlás a provider scope-ra vonatkozóan?

A szolgáltatást mindig a root injector-ban kell megadnia, kivéve, ha azt szeretné, hogy a szolgáltatás csak akkor legyen elérhető, ha egy adott @NgModule-t importál.

Hogyan korlátozhatja a provider scope-ot egy modulra?

Lehetséges a provider scope-ot egy adott modulra korlátozni, ahelyett, hogy a teljes alkalmazás számára elérhetővé tenné. Kétféle módon lehet megtenni.

1.) providedIn a service-ben:

```
import { Injectable } from '@angular/core';
import { SomeModule } from './some.module';

@Injectable({
  providedIn: SomeModule,
})
export class SomeService {
}
```

2.) Deklarálhatjuk a provider-t a service-nek a modul-ban:

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

import { SomeService } from './some.service';

@NgModule({
  providers: [SomeService],
})
export class SomeModule {
}
```

provide a singleton service?

Állítsa a @Injectable () a providedIn tulajdonságát "root" értékre. Ez a szingleton szolgáltatás létrehozásának előnyös módja (az Angular 6.0-tól kezdődően).

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

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

A szolgáltatást vegye fel a root modulba vagy egy olyan modulba, amelyet csak a root modul importál. Az Angular 6.0 előtti szolgáltatások regisztrációjára használták.

```
@NgModule({
  ...
  providers: [MyService],
  ...
})
```

Mi az a megosztott modul (shared module)?

A megosztott modul (shared modul) az a modul, amelybe a gyakran használt directives, pipes, and components egyetlen modulba helyezzük, amelyet az alkalmazáson keresztül megosztunk (importálunk).

Például az alábbi megosztott modul importálja a CommonModule-ot, a FormsModule-ot a közös common directives and components, pipes and directives-nak:

```
import { CommonModule } from '@angular/common';
import { NgModule } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { UserComponent } from './user.component';
import { NewUserDirective } from './new-user.directive';
import { OrdersPipe } from './orders.pipe';

@NgModule({
 imports:      [ CommonModule ],
 declarations: [ UserComponent, NewUserDirective, OrdersPipe ],
 exports:      [ UserComponent, NewUserDirective, OrdersPipe,
                 CommonModule, FormsModule ]
})
export class SharedModule { }
```

Megoszthatom a szolgáltatásokat modulok segítségével?

Nem, nem ajánlott megosztani a szolgáltatásokat a modul importálásával. A megosztott szolgáltatások (services) megszerzésének legjobb módja az „Angular dependency injection”, mivel a modul importálása új szolgáltatás (service) példányt eredményez.

Milyen osztályokat ne adjunk hozzá a deklarációhoz?

1.) Osztály, amelyet már deklaráltak egy másik modulban.

2.) Más modulból importált direktívák.

3.) Modul osztályok.

4.) Service osztályok.

5.) Nem Angular osztályok és objektumok, például karakterláncok, számok, függvények, entitásmodellek, konfigurációk, üzleti logika és segítő osztályok.

Melyek az adatok frissítésének lehetséges forgatókönyvei a változás észleléséhez?

A változás észlelése a következő esetekben működik, amikor az adatok változásainak frissíteniük kell az alkalmazás HTML-jét.

1.) Komponens inicializálása: Az Angular alkalmazás indításakor az Angular elindítja az ApplicationRef.tick () -t, hogy a változás észlelését és a View Rendering-et megcsinálja.

2.) Eseményfigyelő: A DOM eseményfigyelő frissítheti egy Angular komponens adatait, és kiválthatja a változás észlelését is.

```
@Component({
  selector: 'app-event-listener',
  template: `
    <button (click)="onClick()">Click</button>
    {{message}}`
})
export class EventListenerComponent {
  message = '';

  onClick() {
    this.message = 'data updated';
  }
}
```

3.) HTTP Data Request: adatokat egy szerverről a http kérésen keresztül kaphat.

```
data = 'default value';
constructor(private httpClient: HttpClient) {}

  ngOnInit() {
    this.httpClient.get(this.serverUrl).subscribe(response => {
      this.data = response.data; // change detection will happen automatically
    });
  }
```

4.) Macro tasks setTimeout () vagy setInterval (): Az adatokat a setTimeout vagy setInterval callback függvényben frissíthetjük

```
data = 'default value';

  ngOnInit() {
    setTimeout(() => {
      this.data = 'data updated'; // Change detection will happen automatically
    });
  }
```

5.) Micro tasks Promises: a promise callback függvényében frissíthetjük az adatokat.

```
data = 'initial value';

  ngOnInit() {
    Promise.resolve(1).then(v => {
      this.data = v; // Change detection will happen automatically
    });
  }
```

6.) Async műveletek, például Web sockets és Canvas: Az adatok aszinkron módon frissíthetők a WebSocket.onmessage () és a Canvas.toBlob () használatával.

Mi a zone context?

Az Execution Context egy absztrakt fogalom, amely információkat tárol a környezetről az aktuálisan futtatott kódban. A zóna olyan végrehajtási környezetet biztosít, amely az aszinkron műveletek során is fennáll, zóna kontextusnak (zone context) nevezzük. Például a zóna kontextusa azonos lesz a setTimeout callback függvényen kívül és belül is,

```
zone.run(() => {
  // outside zone
  expect(zoneThis).toBe(zone);
  setTimeout(function() {
    // the same outside zone exist here
    expect(zoneThis).toBe(zone);
  });
});
```

Az aktuális zónát a Zone.current segítségével lehet lekérni.

Melyek egy zone lifecycle hooks-jai?

Négy lifecycle hook létezik aszinkron műveletekhez a zone.js fájlból.

1.) onScheduleTask: Ez a hook akkor aktiválódik, amikor új aszinkron feladatot ütemeznek. Például, amikor meghívjuk a setTimeout ()-ot:

```
onScheduleTask: function(delegate, curr, target, task) {
    console.log('new task is scheduled:', task.type, task.source);
    return delegate.scheduleTask(target, task);
  }
```

2.) onInvokeTask: Ez a hook akkor aktiválódik, amikor egy aszinkron feladat végrehajtása előtt áll. Például amikor a setTimeout () callback-re készül.

```
onInvokeTask: function(delegate, curr, target, task, applyThis, applyArgs) {
    console.log('task will be invoked:', task.type, task.source);
    return delegate.invokeTask(target, task, applyThis, applyArgs);
  }
```

3.) onHasTask: Ez a hook akkor váltódik ki, amikor egy zónában belüli egyfajta feladat állapota stabilról (a zónában nincsenek feladatok) instabilra (új feladat kerül ütemezésre a zónában) vagy instabilról stabilra változik.

```
onHasTask: function(delegate, curr, target, hasTaskState) {
    console.log('task state changed in the zone:', hasTaskState);
    return delegate.hasTask(target, hasTaskState);
  }
```

4.) onInvoke: Ez a hook akkor aktiválódik, amikor egy szinkron funkció végrehajtásra kerül a zónában.

```
onInvoke: function(delegate, curr, target, callback, applyThis, applyArgs) {
    console.log('the callback will be invoked:', callback);
    return delegate.invoke(target, callback, applyThis, applyArgs);
  }
```

Milyen metódusokat alkalmaznak az NgZone-t a változás észlelésének ellenőrzésére?

Az NgZone szolgáltatás egy run () metódust biztosít, amely lehetővé teszi egy függvény végrehajtását az Angular zone-on belül. Ezt a funkciót olyan harmadik féltől származó API-k végrehajtására használják, amelyeket a Zone nem kezel, és a változás észlelését automatikusan elindítja a megfelelő időben.

```
export class AppComponent implements OnInit {
  constructor(private ngZone: NgZone) {}
  ngOnInit() {
    // use ngZone.run() to make the asynchronous operation in the angular zone
    this.ngZone.run(() => {
      someNewAsyncAPI(() => {
        // update the data of the component
      });
    });
  }
}
```

Míg a runOutsideAngular () metódus akkor használatos, ha nem akarja elindítani a változás észlelését.

```
export class AppComponent implements OnInit {
  constructor(private ngZone: NgZone) {}
  ngOnInit() {
    // Use this method when you know no data will be updated
    this.ngZone.runOutsideAngular(() => {
      setTimeout(() => {
        // update component data and don't trigger change detection
      });
    });
  }
}
```

Hogyan változtathatja meg a zonejs beállításait?

Megváltoztathatja a zóna beállításait, ha külön fájlba konfigurálja őket, és közvetlenül a zonejs importálása után importálja. Például letilthatja a requestAnimationFrame ()-t, hogy megakadályozza a változás észlelését egyetlen adatfrissítés nélkül, és megakadályozza, hogy a DOM események (egérmozgató vagy görgető esemény) kiváltják a változás észlelését. Tegyük fel, hogy az új zone-flags.js nevű fájl,

```
// disable patching requestAnimationFrame
(window as any).__Zone_disable_requestAnimationFrame = true;

// disable patching specified eventNames
(window as any).__zone_symbol__UNPATCHED_EVENTS = ['scroll', 'mousemove'];
```

A fenti konfigurációs fájl az alábbiak szerint importálható egy polyfill.ts fájlba,

```
/***************************************************************************************************
 * Zone JS is required by default for Angular.
 */
import `./zone-flags`;
import 'zone.js/dist/zone';  // Included with Angular CLI.
```

Hogyan indíthat animációt?

Az Angular egy trigger () függvényt biztosít az animációhoz, hogy összegyűjtse az állapotokat és az átmeneteket egy adott animációnévvel, hogy azt a HTML-sablon kiváltó eleméhez csatolhassa. Ez a funkció figyeli a változásokat, és az eseményindító elindítja a műveleteket, amikor változás történik. Hozzunk létre például egy upDown nevű trigger-t, és csatoljuk a gombelemhez.

```
content_copy
@Component({
  selector: 'app-up-down',
  animations: [
    trigger('upDown', [
      state('up', style({
        height: '200px',
        opacity: 1,
        backgroundColor: 'yellow'
      })),
      state('down', style({
        height: '100px',
        opacity: 0.5,
        backgroundColor: 'green'
      })),
      transition('up => down', [
        animate('1s')
      ]),
      transition('down => up', [
        animate('0.5s')
      ]),
    ]),
  ],
  templateUrl: 'up-down.component.html',
  styleUrls: ['up-down.component.css']
})
export class UpDownComponent {
  isUp = true;

  toggle() {
    this.isUp = !this.isUp;
  }
```

Hogyan konfigurálja az injektorokat különböző szinten a provider-ekkel?

A konfiguráció a három hely egyikén történhet,

1.) Magának a szolgáltatásnak az @Injectable () dekorátorában

2.) A @NgModule () dekorátorban egy NgModule számára

3.) Egy komponens @Component () dekorátorában

Kötelező-e az injectable-t használni minden service osztályban?

Nem. A @Injectable () dekoratőrre nincs szigorúan szükség, ha az osztályon vannak más Angular dekorátorok, vagy nincsenek függőségei. De itt az a fontos, hogy minden olyan osztálythasználja a decorator-t, amelyet injektálnak.

1.) Az alábbi AppService problémamentesen beadható az AppComponent programba. Ennek oka, hogy az AppService-en belül nincsenek függőségi service-ek.

```
export class AppService {
  constructor() {
```

```
    console.log('A new app service');
  }
}
```

2.) Az alábbi AppService dummy decoratorral és httpService problémamentesen beadható az AppComponent programba. Ez azért van, mert a meta információk a dummy decoratorral jönnek létre.

```
function SomeDummyDecorator() {
  return (constructor: Function) => console.log(constructor);
}

@SomeDummyDecorator()
export class AppService {
  constructor(http: HttpService) {
    console.log(http);
  }
}
```

Mi az opcionális függőség (optional dependency)?

Az opcionális függőség (optional dependency) a konstruktor paramétereinél használt paraméter dekorátor, amely a paramétert opcionális függőségként jelöli. Emiatt a DI keretrendszer null-t ad, ha a függőség nem található.

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

constructor(@Optional() private logger?: Logger) {
  if (this.logger) {
    this.logger.log('This is an optional dependency message');
  } else {
    console.log('The logger is not registered');
  }
}
```

Milyen típusú injector hierarchiák vannak?

1.) ModuleInjector hierarchia: Modul szinten konfigurálható egy @NgModule () vagy @ Injectable () segítségével.

2.) ElementInjector hierarchia: implicit módon létrehozta az egyes DOM elemeket. Alapértelmezés szerint üres is, hacsak nem a provider property-ben konfiguráljuk a @Directive () vagy a @Component () címen.

Mik a reactive form-ok?

A reactive form-ok egy modell által vezérelt megközelítés a formok reaktív stílusban történő létrehozásához (az űrlapbemenet idővel változik). Ezek megfigyelhető folyamok köré épülnek, ahol a forma bemenetek és értékek a bemeneti értékek folyamaként vannak megadva. Kövessük az alábbi lépéseket reaktív formok létrehozásához,

1.) Regisztráljuk a reaktív form modult, amely deklarálja a reaktív űrlapokat az alkalmazásban

```
import { ReactiveFormsModule } from '@angular/forms';

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

2.) Hozzon létre egy új FormControl példányt a komponensbe.

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

@Component({
  selector: 'user-profile',
  styleUrls: ['./user-profile.component.css']
})
export class UserProfileComponent {
  userName = new FormControl('');
}
```

3.) Regisztrálja a FormControlt a template-ben

```
<label>
  User name:
  <input type="text" [formControl]="userName">
</label>
```

4.) Végül az alábbiak szerint lesz a reactive form kezelés:

import { Component } from '@angular/core'; import { FormControl } from '@angular/forms';

```
@Component({
  selector: 'user-profile',
  styleUrls: ['./user-profile.component.css']
  template: `
     <label>
       User name:
       <input type="text" [formControl]="userName">
     </label>
  `
})
export class UserProfileComponent {
  userName = new FormControl('');
}
```

Mik a dinamikus formok?

A dinamikus form egy olyan minta, amelyben metaadatokon alapuló dinamikusan építünk egy űrlapot, amely leír egy üzleti objektum modellt. Létrehozhatja őket reaktív form alapján.

Mik azok a template driven formok?

A template driven formok olyan modell által vezérelt formok, amelyekbe a logikát, a validációt, a vezérlőket stb. írjuk a kód sablonrészébe direktívák segítségével. Alkalmasak egyszerű logikához, és

kétirányú kötést (two-way binding) használnak a [(ngModel)] szintaxissal. Például könnyen létrehozhat regisztrációs űrlapot az alábbi egyszerű lépések végrehajtásával:

1.) Importálja a FormsModule-t

```
import { BrowserModule } from '@angular/platform-browser';
   import { NgModule } from '@angular/core';
   import {FormsModule} from '@angular/forms'
   import { RegisterComponent } from './app.component';
   @NgModule({
     declarations: [
       RegisterComponent,
     ],
     imports: [
       BrowserModule,
       FormsModule
     ],
     providers: [],
     bootstrap: [RegisterComponent]
   })
   export class AppModule { }
```

2.) Az ngModel szintaxissal a form template-et a komponenssel összekötjük:

```
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name">
```

3.) Csatolja az NgForm direktívát a tag-hez a FormControl példányok létrehozása és regisztrálása érdekében

```
<form #registerForm="ngForm">
```

4.) Alkalmazza a validációsüzenetet a form control-hoz

```
<label for="name">Name</label>
<input type="text" class="form-control" id="name"
       required
       [(ngModel)]="model.name" name="name"
       #name="ngModel">
<div [hidden]="name.valid || name.pristine"
     class="alert alert-danger">
  Please enter your name
</div>
```

5.) Küldjük el az űrlapot az ngSubmit direktívával, és az űrlap elküldéséhez indítsuk el az űrlap alján a type = "submit" gombot.

```
<form (ngSubmit)="onSubmit()" #heroForm="ngForm">
// Form goes here
<button type="submit" class="btn btn-success" [disabled]="!registerForm.form.valid">Submit</button>
```

A teljes form így fog kinézni:

```
```html
<div class="container">
    <h1>Registration Form</h1>
    <form (ngSubmit)="onSubmit()" #registerForm="ngForm">
```
```

```
      <div class="form-group">
        <label for="name">Name</label>
        <input type="text" class="form-control" id="name"
               required
               [(ngModel)]="model.name" name="name"
               #name="ngModel">
        <div [hidden]="name.valid || name.pristine"
             class="alert alert-danger">
          Please enter your name
        </div>
      </div>
           <button type="submit" class="btn btn-success"
[disabled]="!registerForm.form.valid">Submit</button>
    </form>
</div>
```

Mi a különbség a reactive form és a template driven form között?

Feature	Reactive	Template-Driven
Form model setup	A komponensben expliciten létrehozva (FormControl példány)	Direktívák hozzák létre
Adatfrissítések	Szinkron	Aszinkron
Űrlap egyéni validálás	Függvényként definiálva	Direktívák határozzák meg
Tesztelés	Nincs kölcsönhatás a változás észlelési ciklussal	Szüksége van a változás észlelési folyamat ismeretére
Változtathatóság	Immutable (mindig új értéket adunk vissza a FormControl példányhoz)	Mutable (a tulajdonság mindig új értékre módosul)
Méretezhetőség	Skálázható alacsony szintű API-k használatával	Kevésbé skálázható az API-k absztrakciója miatt

Milyen módon lehet csoportosítani az form control-okat?

1.) FormGroup: Meghatározza az űrlapot egy fix vezérlővel, amelyeket egy objektumban együtt lehet kezelni. A FormControl példányhoz hasonló tulajdonságokkal és módszerekkel rendelkezik. Ez a FormGroup beágyazható összetett űrlapok létrehozására az alábbiak szerint.

```
import { Component } from '@angular/core';
import { FormGroup, FormControl } from '@angular/forms';

@Component({
  selector: 'user-profile',
  templateUrl: './user-profile.component.html',
  styleUrls: ['./user-profile.component.css']
})
export class UserProfileComponent {
  userProfile = new FormGroup({
    firstName: new FormControl(''),
    lastName: new FormControl(''),
    address: new FormGroup({
          street: new FormControl(''),
          city: new FormControl(''),
          state: new FormControl(''),
          zip: new FormControl('')
        })
```

```
  });

  onSubmit() {
    // Store this.userProfile.value in DB
  }
}
<form [formGroup]="userProfile" (ngSubmit)="onSubmit()">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

  <div formGroupName="address">
    <h3>Address</h3>

    <label>
      Street:
      <input type="text" formControlName="street">
    </label>

    <label>
      City:
      <input type="text" formControlName="city">
    </label>

    <label>
      State:
      <input type="text" formControlName="state">
    </label>

    <label>
      Zip Code:
      <input type="text" formControlName="zip">
    </label>
   </div>
    <button type="submit" [disabled]="!userProfile.valid">Submit</button>

</form>
```

2.) FormArray: Dinamikus formot határoz meg tömb formátumban, ahol futás közben hozzáadhatja és eltávolíthatja a vezérlőket. Ez akkor hasznos a dinamikus űrlapoknál, ha nem tudja, hány vezérlő lesz jelen a csoporton belül.

```
import { Component } from '@angular/core';
 import { FormArray, FormControl } from '@angular/forms';

 @Component({
   selector: 'order-form',
   templateUrl: './order-form.component.html',
   styleUrls: ['./order-form.component.css']
 })
 export class OrderFormComponent {
   constructor () {
     this.orderForm = new FormGroup({
       firstName: new FormControl('John', Validators.minLength(3)),
       lastName: new FormControl('Rodson'),
       items: new FormArray([
         new FormControl(null)
       ])
     });
   }

   onSubmitForm () {
     // Save the items this.orderForm.value in DB
```

```
   }

   onAddItem () {
     this.orderForm.controls
     .items.push(new FormControl(null));
   }

   onRemoveItem (index) {
     this.orderForm.controls['items'].removeAt(index);
   }
 }
<form [formControlName]="orderForm" (ngSubmit)="onSubmit()">

  <label>
    First Name:
    <input type="text" formControlName="firstName">
  </label>

  <label>
    Last Name:
    <input type="text" formControlName="lastName">
  </label>

  <div>
  <p>Add items</p>
  <ul formArrayName="items">
    <li *ngFor="let item of orderForm.controls.items.controls; let i = index">
      <input type="text" formControlName="{{i}}">
      <button type="button" title="Remove Item" (click)="onRemoveItem(i)">Remove</button>
    </li>
  </ul>
  <button type="button" (click)="onAddItem">
    Add an item
  </button>
 </div>
```

Hogyan frissítheti az form model bizonyos tulajdonságait?

A patchValue () metódussal frissítheti a form model-ben meghatározott specifikus tulajdonságokat.

```
updateProfile() {
  this.userProfile.patchValue({
    firstName: 'John',
    address: {
      street: '98 Crescent Street'
    }
  });
}

  <button (click)="updateProfile()">Update Profile</button>
```

Mi a célja a FormBuildernek?

A FormBuilder syntactic sugar-ként használható a FormControl, FormGroup vagy FormArray példányok egyszerű létrehozásához. Ez hasznos a komplex reaktív formok elkészítéséhez szükséges kód mennyiségének csökkentésében. A @ angular / forms csomag injectable segédosztálya.

```
export class UserProfileComponent {
  profileForm = this.formBuilder.group({
```

```
    firstName: [''],
    lastName: [''],
    address: this.formBuilder.group({
      street: [''],
      city: [''],
      state: [''],
      zip: ['']
    }),
  });
  constructor(private formBuilder: FormBuilder) { }
  }
```

Hogyan ellenőrizheti az űrlapok modellváltozásait?

Hozzáadhat egy getter tulajdonságot a komponensen belül, hogy a modell JSON-ábrázolását adja vissza a fejlesztés során. Ez hasznos annak ellenőrzéséhez, hogy az értékek valóban a beviteli mezőből a modellbe áramlanak-e, és fordítva.

```
export class UserProfileComponent {

  model = new User('John', 29, 'Writer');

  // TODO: Remove after the verification
  get diagnostic() { return JSON.stringify(this.model); }
}
```

és adjon hozzá diagnostic binding-ot az űrlap teteje közelében

```
{{diagnostic}}
<div class="form-group">
  // FormControls goes here
</div>
```

Mik a state CSS osztályok, amelyeket az ngModel nyújt?

Az ngModel direktíva frissíti a form control-t speciális Angular CSS osztályokkal annak állapotának tükrözése érdekében.

Form control state	If true	If false
Visited	ng-touched	ng-untouched
Value has changed	ng-dirty	ng-pristine
Value is valid	ng-valid	ng-invalid

Hogyan reset-elhetjük a form-ot?

model-driven form-ban az űrlapot csak az űrlapmodellünkön a reset () függvény meghívásával állíthatjuk vissza. Például visszaállíthatja az form beküldésekor az alábbiak szerint:

```
onSubmit() {
  if (this.myform.valid) {
    console.log("Form is submitted");
    // Perform business logic here
    this.myform.reset();
  }
}
```

Milyen típusú validator függvények vannak?

Reaktív formokban a validátorok lehetnek szinkron vagy aszinkron függvények,

1.) Sync validators: Ezek azok a szinkron függvények, amelyek egy vezérlőpéldányt vesznek fel, és azonnal visszaadnak vagy validációs hibákat, vagy null értéket. Ezenkívül ezek a függvények második argumentumként adódtak át, miközben az űrlapvezérlést példányosították. A fő használati esetek egyszerű ellenőrzések, például, hogy egy mező üres-e, túllépi-e a maximális hosszúságot stb.

2.) Async validators: Ezek azok az aszinkron függvények, amelyek egy vezérlőpéldányt vesznek fel, és olyan Promise-t vagy Observable-t adnak vissza, amely később érvényesítési hibák vagy null értékeket ad ki. Ezenkívül ezek a függvények második argumentumként adódtak át, miközben az űrlapvezérlést példányosították. A fő felhasználási esetek összetett ellenőrzések, például egy szerver elérése a felhasználónév vagy az e-mail elérhetőségének ellenőrzésére.

```
this.myForm = formBuilder.group({
    firstName: ['value'],
    lastName: ['value', *Some Sync validation function*],
    email: ['value', *Some validation function*, *Some asynchronous validation function*]
});
```

Mondanál példát a beépített validátorokra?

Reaktív formokban pl. required és minlength beépített validátor:

```
this.registrationForm = new FormGroup({
    'name': new FormControl(this.hero.name, [
      Validators.required,
      Validators.minLength(4),
    ])
  });
```

Míg template-driven form-nál mind a required mind a minlength elérhetőek attribútumként.

Hogyan optimalizálja az async validator-ok teljesítményét?

Mivel minden validator minden form érték-változtatás után fut, az aszinkron ellenőrzőkkel jelentős hatást gyakorol a teljesítményre azáltal, hogy minden billentyűleütésnél lefut a külső API-t. Ezt a helyzetet úgy lehet elkerülni, hogy az űrlap érvényességét késleltetjük az updateOn tulajdonság módosításával (alapértelmezett).

1.) Template-driven forms: ngModelOptions direktívát használva:

```
<input [(ngModel)]="name" [ngModelOptions]="{updateOn: 'blur'}">
```

2.) Reactive-forms: Set the property on FormControl instance

```
name = new FormControl('', {updateOn: 'blur'});
```

Mi a host property a css-ben?

A :host pseudo-class selector olyan elemek stílusának beállítására szolgál, amelyek tartalmazzák az adott elemet. Egy szülő elemhez egy szegélyt adunk:

```
//Other styles for app.component.css
//...
:host {
  display: block;
  border: 1px solid black;
  padding: 20px;
}
```

Hogyan érheti el az aktuális útvonalat?

A router URL property-jével:

1.) Importálja a routert a @ angular / router ből

```
import { Router } from '@angular/router';
```

2.) Inject router inside constructor

```
constructor(private router: Router ) {

}
```

3.) Access url parameter

```
console.log(this.router.url); //  /routename
```