**1. Getting Started**

4. Angular vs Angular 2 vs Latest Angular Version

პირველად შექმნეს**AngularJS(Angular 1)**, რომელიც დაბალი პერფორმანსით გამოირჩეოდა და დიდ აპლიკაციებზე გარკვეული ბაგები ჰქონდა.

2016 წლეს სრულიად გადაწერეს AngularJS (რათა მასში არსებული ბაგები გამოესწორებინათ) და შედეგად გამოვიდა - **Angular 2,**რომლის შემდეგ ვერსიებშიც ასე რადიკალური ცვლილება არ განხორციელებულა, მხოლოდ დეტალები ახლდებოდა.

Angular-ის ვერსიები 2-ის შემდეგ დღემდე გამოდის ახალი, მხოლოდ მესამე (**Angular 3**) არ არსებობს.

*დამატებით:*

npm-ის განახლება: npm install -g npm

CLI-ის განახლება:

npm uninstall -g angular-cli @angular/cli

npm cache verify

npm install -g @angular/cli

**1. Getting Started**

6. Project Setup and First App

**ანგულარის დაინსტალირების ინსტრუქცია:**

1. NodeJS უნდა ჩამოვწერი მისი საიტიდან და დავაინსტალირო

2. შემდეგ შემდეგი ბრძანება უნდა გავუშვა ქომანდ ლაინში:

[sudo](მაკში ან ლინუქსში) npm install -g @angular/cli@latest

3. ng new [Project Name] --no-strict (გათიშავს strict mode-ს) *(სახელი არ უნდა შეიცავდეს ცარიელ სივრცეს ან სიტყვა test-ს)*

4. შემდეგ ვირჩევთ სასურველ მოდებს

5. ng serve -o  პროექტის ლაივში გასაშვებად (ფოლდერში უნდა იყოს cli).

**1. Getting Started**

7. Editing the First App

**ანგულარის ფაილური სისტემა:**

# package.json - dependency folder - დამოკიდებულებების ფაილი.

# მთავარი ფოლდერია **app,**სადაც ვმუშაობთ ჩვენს აპლიკაციაში.

# ანგულარი მუშაობს კომპონენტებით, რომელსაც აქვს .ts, .html, .css ფაილები.

# index.html არის მთავარი html, რომელშიც მიდის სხვა დანარჩენი კომპონენტები

**1. Getting Started**

12. A Basic Project Setup using Bootstrap for Styling

**Bootstrap-ის დაინსტალირება (ან სხვა ექსტერნალ ფრეიმვორკის):**

CLI-ის ბრძანებით: npm install --save (ლოკალურად აინსტალირებს) bootstrap@latest

შემდეგ angular.json ფაილში უნდა მივუთითოთ, ამ შემთხვევაში bootstrap-ის css-ის მისამართი, styles-ის მასივში: node\_modules/bootstrap/dist/css/bootstrap.min.css

შემოწმება:

ბრაუზერის ინსპექტში **sources**-ში **styles.css**-ში წერია.

**StrictMode-ის გათიშვა** შეიძლება:**tsconfig.json**-ფაილში - strict: false

**2. The Basics**

15. How an Angular App gets Loaded and Started

index.html ფაილი იკითხება, მასში ჩასმული <app-root></app-root>-ის საშუალებით კი სხვა დანარჩენი კომპონენტები.

<app-root>Loading...</app-root>-Loading წაიკითხება, ოღონდ მანამ, სანამ თავად კომპონენტი დარენდერდება

როდესაც ლაივზე ვუშვებთ, ng serve ბილდავს აპლიკაციას და ამატებს ჯავასკრიპტის იმპორტებს ბოლოში, რომლებიც წაიკითხავენ კომპონენტებს.

**main.ts**ფაილი იკითხება პირველად **index.html**-თან ერთად:

platformBrowserDynamic().bootstrapModule(AppModule) უთითებს კოდს რომ დაარენდეროს ყველა კომპონენტი, რომელიც **appModule**-ში შედის, იქ კი **bootstrap**-ის მასივშია მითითებული **AppComponent**-ი, რომელიც უნდა გაუშვას.

**2. The Basics**

16. Components are Important!

**კომპონენტები:**

მთავარი კომპონენტია *(root component)* **app.component**-ი.

მასში შენდება სხვა კომპონენტები.

თითეულ კომპონენტს აქვს მისი html, css და ბიზნეს ლოგიკა.

**2. The Basics**

17. Creating a New Component

**კომპონენტის შექმნა (მანუალურად) #1:**

1. სასურველია ჰქონდეს საკუთარი ფოლდერი app ფოლდერში.

2. შემდეგ ვქმნით **ts** ფაილს სტანდარტიზებული დაბოლოების მქონე სახელით, მაგალითად: **server.component.ts.**

3. შემდეგ შიგნით იქმნება კლასი და იგი ექსპორტდება, სახელი კი მსგავსი უნდა ჰქონდეს, რაც ფაილს ჰქვია:

1. export class ServerComponent {
2. }

4. იმისათვის, რომ ფაილი კომპონენტად იყოს აღქმული, კომპონენტის დეკორატორი უნდა შემოვიტანოთ:

1. import { Component } from "@angular/core"
3. @Component({
5. })
6. export class ServerComponent {
7. }

5. შემდეგ კი უნდა ვადაკონფიგურიროთ კომპონენტის დეკორატორში ჩანესტებული ობიექტი:

1. import { Component } from "@angular/core"
3. @Component({
4. selector: 'app-server',//სახელი, რომლითაც კომპონენტს გამოვიძახებთ. უნდა იყოს უნიკალური
5. templateURL: './server.component.html', //html თემფლეითი, რომელიც მიებმება ჩვენს კომპონენტს
6. })
7. export class ServerComponent {
8. }

**2. The Basics**

18. Understanding the Role of AppModule and Component Declaration

**კომპონენტის შექმნა (მანუალურად) #2:**

ანგულარი **მოდულს  (მაგ. app.module.ts)**იყენებს იმისათვის, რომ ყველა კომპონენტი, დირექტივი, სერვისი და ა.შ. შეკრას პაკეტებად, ანუ გააერთიანოს ყველაფერი ერთ სივრცეში.

მოდული იყენებს NgModule-ს, რომელშიც ინესტება ობიექტი, რაშიც შედის: declarations, imports, providers და bootstrap მასივები.

**imports**- საშუალებას გვაძლევ ამ მოდულში სხვა მოდულები დავამატოთ.

**providers**- ძირითადად სერვისების დასაიმპორტებლად,

აქედან, კომპონენტები დეკლარირდება declarations-ში:

1. import { AppComponent } from './app.component'
2. import { ServerComponent } from './server/server.component'
3. import { BrowserModule } from '@angular/platform-browser'
5. @NgModule({
6. declarations: [
7. AppComponent,
8. ServerComponent
9. ]
10. imports: [
11. BrowserModule
12. ],
13. providers: [],
14. bootstrap: [AppComponent]
15. })
16. export class AppModule{}

**2. The Basics**

19. Using Custom Components

**კომპონენტის შექმნა (მანუალურად) #2:**

ამ შექმნილ კომპონენტების გამოყენება კი თაგის სტილში შეიძლება იმ სახელით, რომელიც **selector**-ში დაერქვა, მაგალითად service კომპონენტის გამოყენება **app.component.html**-ში:

1. <h3>I'm in the AppComponent!</h3>
2. <hr>
3. <app-server></app-server>

**2. The Basics**

20. Creating Components with the CLI & Nesting Components

**კომპონენტის შექმნა (ავტომატურად) #1:**

ვიყენებთ **cli**ბრძანებას: ng generate component [component-name] ან მოკლედ: ng g c [component-name]

იგი**ავტომატურად დააიმპორტებს app.module.ts**-ში

**2. The Basics**

21. Working with Component Templates

**კომპონენტში მუშაობა:**

შეგვიძლია **templateURL**-ის ნაცვლად დავწეროთ - **template**და html-კოდი იქვე დავწეროთ, მაგალითად:

1. @Component({
2. selector: 'app-servers',
3. template: "
4. <h1>servers</h1>
5. <app-server></app-server>
6. ",
7. styleUrls:['./servers.component.css'] //აქ კავშირდება სტილი
8. })

**2. The Basics**

22. Working with Component Styles

**კომპონენტში მუშაობა:**

შეგვიძლია **styleUrls**-ის მასივში სხვა css ფაილების დაკავშირებაც, ან იქვე დაწერა, ოღონდ styleUrls-ის ნაცვლად **styles**უნდა დავწეროთ და:

1. @Component({
2. selector: 'app-servers',
3. templateUrl: "./servers.component.html",
4. styles:[`
5. h3 {
6. color: blue
7. }
8. `]
9. })

**2. The Basics**

23. Fully Understanding the Component Selector

**კომპონენტში მუშაობა (სელექტორები):**

კომპონენტის სელექტორის შექმნა **css**ჰგავს, მაგალითად: selector: 'app-service' საშუალებას გვაძლევს **კომპონენტი თაგის სტილში** გამოვიძახოთ სხვა კომპონენტში (<app-service></app-service>),

თუმცა ასევე შესაძლებელია მისი **ატრიბუტული სელექტორის** სტილშიც გამოძახება, მაგალითად: selector: '[app-service]' ასე დაიწერება და **html**-ში გამოიძახება შემდეგნაირად: <div app-servers></div>

კიდევ ერთი ალტერნატივაა **კლასის სტილში**გამოძახება: selector: '.app-service' , **html**-ში კი შემდეგნაირად წავა: <div class="app-service"></div>

**2. The Basics**

25. What is Databinding?

**მონაცემთა ბაინდინგი (Databinding):**

იგი წარმოადგენს კომუნიკაციას **typescript**-სა და **html**თემფლეითს შორის.

იგი მოიცავს რამდენიმე ტიპის **კომუნიკაციას**:

*Output Data:*

1. **String Interpolation** - {{ data }}

2. **Property Binding** - [property]="data"

*React to (User) Events:*

1. **Event Binding**- (event)="expression"

Combination of both: **Two-Way-Binding** - [( ngModel)]="data"

**მონაცემთა ბაინდინგი (Databinding) String Interpolation:**

**ts** ფაილში:

1. export class ServerComponent {
2. serverID: number = 10;
3. }

**html** ფაილში:

<h1>Id is {{ serverID }} </h1>

*სტრინგის ინტერპოლაციის გამოყენება****შემდეგნაირადაც შეიძლება:***

<h1>Name is {{ 'Server' }} </h1>

ასევე შესაძლებელია მეთოდის გამოძახებაც, მაგალითი:

**ts** ფაილში:

1. export class ServerComponent {
2. serverID: number = 10;
4. getId(){
5. return this.serverID
6. }
8. }

**html** ფაილში:

<h1>Name is {{ getId() }} </h1>

**2. The Basics**

27. Property Binding

**მონაცემთა ბაინდინგი (Databinding) Property Binding:**

html-ის ყველა ფროფერთის ბაინდინგია შესაძლებელი (და არა მხოლოდ), მათი კვადრატული ფრჩხილებში ჩასმისა და შესაბამის ექსპრეშენთან გატოლების საშუალებით, რომელიც სამომავლოდ მის სტეიტს განსაზღვრავს, მაგალითად:

**html:**

1. <button disabled> Add </button> //სტანდარტული
2. <button [disabled]="add"> Add </button> //სკრიპტთან დაკავშირებული, ანუ დაბაინდებული

**ts:**

1. export class ServerComponent {
2. add: boolean = true //ან false
3. }

**2. The Basics**

28. Property Binding vs String Interpolation

**მონაცემთა ბაინდინგი (Databinding) Property Binding vs String interpolation:**

სტრინგის ინტერფოლაციის ფროფერთი ბაინდინგით ჩანაცვლება შეიძლება შემდეგნაირად (**innerText**):

**html:**

1. <p > {{ server }} </p> //ჩანაცვლებამდე
2. <p [innerText]="server "></p> //ჩანაცვლების შემდეგ

**ts:**

1. export class ServerComponent {
2. server: string= 'Hello, Man'
3. }

*როდის რომელი უნდა გამოვიყენოთ:*

თუ თემფლეითში ტექსტის აუთფუთი გვინდა, მაშინ სტრინგ ინტერფოლაცია, ხოლო თუ ელემენტების შეცვლა გვსურს - ფროფერთი ბაინდინგი.

**2. The Basics**

29. Event Binding

**მონაცემთა ბაინდინგი (Databinding) Event Binding:**

იგი იყენებს ფრჩხილებს და ექსპრეშენთან გატოლებას. მაგალითად:

**html:**

1. <button (click)="onCreateServer()" > Create </button>
2. <h1>{{ server }} </h1>

**ts:**

1. export class ServerComponent {
2. server: string;
4. onCreateServer(){
5. this.server = "created done"
6. }
7. }

**მონაცემთა ბაინდინგი (Databinding) Event Binding - Event:**

**$event** - ჩაეწოდება მეთოდს არგუმენტად, იგი მოიცავს ინფორმაციას მიმაგრებულ თაგზე. იგი ასევე მოიცავს target ინფორმაციას, რომელშიც შედის ფროფერთი value, რომელიც ინახავს მნიშვნელობას, რომელიც მაგალითად ინფუთშია შეყვანილი.

**HTMLInputElement**-ით ტაიპსკრიპტს ვატყობინებთ, რომ ამ ელემენტის ტიპი ინფუთ ტიპს წარმოადგენს.

**html:**

1. <input type="text" (input)="updating($event)"/>
2. <h1>{{ info }} </h1>

**ts:**

1. export class ServerComponent {
2. info: string;
3. updating(event: Event){
4. this.info = (<HTMLInputElement>event.target).value
5. }
6. }

**2. The Basics**

33. Two-Way-Databinding

**მონაცემთა ბაინდინგი (Databinding) Two-Way-Binding:**

იმისათვის, რომ თუ-ვეი-ბაინდინგით ვისარგებლოთ, ჯერ აღნიშნული მოდული მონდა დავაიმპორტოთ ჩვენი აპის მოდულში: import { FormsModule } from '@angular/forms';

წინაზე **$event**-ის საშუალებით მივწვდით ინფუთს, თუ-ვეი-ბაინდინგი კი საშუალებას გვაძლევს უფრო მარტივად მივწვდეთ ელემენტის მნიშნვნელობას, ამისთვის  ალემენტს ატრიბუტად უნდა ჩავუმატოთ შემდეგი დირექტივი: [(ngModel)]="serverName", იგი ორმაგად განაახლებს ინფორმაციას, მაგალითში:

**html:**

1. <input [(ngModel)]="serverName" />
2. <h1>{{ serverName }} </h1>

**ts:**

1. export class ServerComponent {
2. serverName: string;
3. }

**2. The Basics**

36. Understanding Directives

**დირექტეივები (Directives) ზოგადი:**

ისინი DOM-ში არსებული ინსტრუქციებია. კომპონენტებიც თავისთავად დირექტეივებს წარმოადგენენ.

მაგალითად:

<p appTurnGreen>Recives a green background!</p>

იგი დირექტივის ts ფაილში შემდეგნაირად იქნებოდა დეკლარირებული:

1. @Directive({
2. selector: '[appTurnGreen]'
3. })
4. export class TurnGreenDirective {
5. ...
6. }

**დირექტეივები (Directives) - \*ngIf დირექტივი:**

იგი წარმოადგენს ქონდიშენალ დირექტივს. \* მიუთითებს იმაზე, რომ იგი სტრუქტურულ დირექტივს წარმოადგენს.

**html:**

<p \*ngIf="server" > Server name is {{ server }} </p>

**2. The Basics**

38. Enhancing ngIf with an Else Condition

**დირექტეივები (Directives) - \*ngIf & else:**

**else**ქონდიშენის გამოყენება შემდეგნაირადაა შესაძლებელი:

1. <p \*ngIf="name; else noServer">Appear Conditionally </p>
2. <p #noServer> No Appear </p>

**noServer** ამ შემთხვევაში წარმოადგენს ლოქალ რეფერენსს - **#noServer.**

**2. The Basics**

39. Styling Elements Dynamically with ngStyle

**დირექტეივები (Directives) Attribute:**

სტრუქტურული დირექტივებისგან განსხვავებით, ატრიბუტული დირექტივები არ ამატებენ ან შლიან ელემენტებს. ისინი უბრალოდ ცვლიან ელემენტს ადგილზევე.

ერთ-ერთი ჩაშენებული ატრიბუტული დირექტივებია: **ngStyle**და **ngClass**. ისინი კვადრატულ ფრჩხილებში სხდებიან, რათა ფროფერთი ბაინდინგი მოხდეს, ანუ ლოგიკის ფაილიდან წამოიღოს ინფორმაციამის, რაც მის სტეიტს განსაზღვრავს:

**ngStyle**- ადგილზევე იწერება **css**სტილი,

**ngClass**- ხდება კლასის მიმაგრება.

ორივეში შიეძლება ქონდიშენალ ფორმის გამოყენება. **ngStyle**მაგალითები:

**სტანდარტულად:**

<p [ngStyle]="{backgroundColor: 'blue' }"> Hello </p>

**ან**

<p [ngStyle]="{'background-color': 'blue' }"> Hello </p>

**2. The Basics**

39. Styling Elements Dynamically with ngStyle

**ლოგიკაზე დამოკიდებულად:**

<p [ngStyle]="{backgroundColor: getColor()}"> Hello </p>

**ts:**

1. export class ServerComponent {
2. getColor(){
3. return 'blue'
4. }
5. }

**2. The Basics**

40. Applying CSS Classes Dynamically with ngClass

**დირექტეივები (Directives) Attribute:**

**ngClass**-ის შემთხვევაში, მაგალითად გვაქვს **.color კლასი css**-ში ან ლოგიკის **style**-ში დეკლარირებული:

**html:**

<p [ngClass]="{color: serverStatus === 'online' }" > Enabled Class </p>

**ts:**

1. export class ServerComponent {
2. serverStatus: string = 'online'
3. }

**2. The Basics**

41. Outputting Lists with ngFor

**დირექტეივები (Directives) Structural:**

არსებობს **\*ngFor** დირექტივი, რომელიც ლუპს აკეთებს და გამოაქვს ინფორმაცია. მაგალითად:

**html:**

<p \*ngFor=" let name of names " > {{ name }} </p>

**ts:**

1. export class ServerComponent {
2. names: string[] = ['გიორგი', 'ბექა', 'მარი', 'გოგა']
3. }

**2. The Basics**

43. Getting the Index when using ngFor

**დირექტეივები (Directives) Structural:**

**\*ngFor**-ს მნიშვნელობის გარდა შეუძლია წამოიღოს ინდექსი, ანუ კონტენტის **ინდექსირებული პოზიცია**, ამისთვის მას უნდა დავამატოთ გამოვტოთ ;-ით და დავწეროთ შემდეგნაირი სინტაქსი: let i = index , მაგალითად:

<p \*ngFor="let name of names: let i = index"> {{ name }} ინდექსი აქვს {{ i }}</p>

**4. Debugging**

61. Understanding Angular Error Messages

**Debugging:**

ერორის მესიჯზე გაცნობისას უნდა გავითვალისწინოთ, რომ შეიძლება ისეთ პოზიციები მოგვაწოდოს, რომლებიც ალოგიკურია, რადგან იგი კითხულობს უკვე დაბილდულ აპლიკაციას, რომელშიც პოზიციები შეცვლილია.

**4. Debugging**

62. Debugging Code in the Browser Using Sourcemaps

**Debugging:**

ბრაუზერის დეველოპერ თულსში შესვლისას Source-ში უნდა გადავიდეთ და ვნახოთ **main.bundle.js**, სადაც ჩვენი პროექტი ინახება. მანდ უნდა მოვნახოთ ერორში მითითებული ლოგიკა და შემდეგ ლაინის ნომერზე დაჭერისას გადაგვიყვანს **ts**ფაილზე, რადგან აღნიშნული **js**ფაილები მხარს უჭერენ **sourceMap**-ს, რომელიც ემატება **cli**-ს საშუალებით js ფაილებს, და იგი საშუალებას აძლევს ბრაუზერს js გადათარგმნოს ts-ში.

თუმცა, იმ შემთხვევაში, თუ დიდი კოდი გვაქვს, ბანდლში ერორის ნახვა რთულია, ამიტომაც პირდაპირ შეგვიძლია ბრაუზერიდან ts ფაილში გადავიდეთ შემდეგნაირად: **source/webpack/./src/app** და აქაა ყველა ფაილი

**5. Components & Databinding Deep Dive**

64. Splitting Apps into Components

**კომონენტის გენერირებისას:**

თუ გვუსრს **ტესტინგ**ფაილი არ ჰქონდეს ჩვენს კომპონენტს, ავტომატურად შემდეგნაირად უნდა დავაგენერიროთ:

ng g c --skipTests=true

**5. Components & Databinding Deep Dive**

66. Binding to Custom Properties

**Costum properties:**

ყველა კომპონენტის ფროფერთი ხელმისაწვდომია მხოლოდ შიგნით კომპონენტში. თუმცა, შესაძლებალია ფროფერთიები სხვა კომპონენტებისთვისაც გავხადოთ ხელმისაწვდომი, ამისთვის გამოიყენება დეკორატორი **@Input()**, რომელიც ასევე იმპორტირებული უნდა იყოს **@angular/core**-დან. მაგალითად:

***[აქ იწერება შვილობილის მიმღები ფროფერთი]="აქ იწერება მშობელში დეკლარირებული ან გაგზავნილი ფროფერთი/მეთოდი"***

მშობელი კომპონენტის **ts:**

1. export class ServerComponent {
2. serverParent: string = 'მე ვარ მშობლის იფნორმაცია'
3. }

მშობელი კომპონენტის **html:**

<app-server [server]="serverParent"></app-server>

შვილობილის **ts:**

1. export class ServerComponent {
2. @Input() server: string;
3. }

**5. Components & Databinding Deep Dive**

67. Assigning an Alias to Custom Properties

**Costum properties (Alias):**

შვილობილ კომპონენტს ასევე შეუძლია **სხვა სახელწოდებით**მიაწოდოს მიმღები ფროფერთი, შემდეგნაირად:

**html:**

<app-server [diffName]="serverParent"></app-server>

**ts:**

@Input('diffName') element:string;

**5. Components & Databinding Deep Dive**

68. Binding to Custom Events

**Costum properties - Costum events:**

ამ შემთხვევაში, წინასგან განსხვავებით, ფროფერთი შეგვიძლია ხელმისაწვდომი გავხადოთ შვილობილიდან მშბლისთვის (წინაზე პირიქით იყო - მშობლელი აწვდიდა შვილობილს). აღნიშნულისთვის გამოიყენება ევენთ ბაინდინგი, რომელიც მოუსმენს ცვლილებას, მაგალითად:

მშობლის html:

<app-server (serverCreated)="onServerAdded($event)" ></app-server>

**$event** - საშუალებას გვაძლევს კომპონენტთან დაკავშირებული ინფორმაცია დავიჭიროთ

**მშობლის ts:**

1. dataGot:string;
3. onServerAdded(data: string){
4. this.dataGot = data;
5. }

**შვილობილის ts:**

1. @Output() serverCreated = new EventEmitter<string>()
2. onAdd(){
3. this.serverCreated.emit('Giorgi')
4. }

**EventEmitter**წარმოადგენს ჯენერიკ ტიპს, ამიტომაც ვიყენებთ **<>** - ამ ნიშნებს. იგი იმპორტირებული უნდა იქნეს**@angular/core**-დან.

**@Output**-იც იმპორტირებული უნდა იყოს **@angular/core**-დან, აქვს alies-იც.

**5. Components & Databinding Deep Dive**

71. Understanding View Encapsulation

**View Encapsulation:**

ანგულარი კომპონენტებს აიძულებს დამოუკიდებელი css პარამეტრები ჰქონდეთ, რაც ხდება encapsulation ტაქტიკის გამოყენებით, რომელიც ელემენტებს ატრიბუტებს ამატებს და განუსაზღვრავს მისთვის განკუთვნილ სტილებს.

იგი ამ ქმედებით shadowDOM-ს ბაძავს, რომელსაც სამწუხაროდ ყველა ბრაუზერი არ უჭერს მხარს, სადაც თითოეულ ელემენტს აქვს მისი საკუთარი shadowDOM.

**5. Components & Databinding Deep Dive**

72. More on View Encapsulation

**View Encapsulation (use):**

ჩვენ შეგვიძლია **Encapsulation**-ით მაინპულირება, ამისთვის კონკრეტულ კომპონენტის Component დეკორატორში უნდა ჩავამატოთ ფროფერთი: **encapsulation**: **ViewEncapsulation**, რომელიც უნდა დავაიმპორტოთ **@angular/core**-დან.

მას აქვს **3 ოფშენი**:

1. **Emulated**- დეფაულტზე აყენია

2. **none**- გათიშავს მას (ანუ ამ კომპონენტის css-ში დეკლარირებული სტილი ყველა კომპონენტზე გავრცელდება)

3. **ShadowDom**- იყენებს აღნიშნულ ტექნოლოგიას, რომელიც იგივე შედეგს იძლევა, როგორსაც Emulated, თუმცა იმ ბრაუზერებში, რომლებიც მას მხარს უჭერენ.

**5. Components & Databinding Deep Dive**

73. Using Local References in Templates

**Local References:**

**local reference** შეგვიძლია ნებისმიერ ელემენტს მივამაგროთ, იგი იწყება **#**-ით და შემდეგ სასურველი რამ. ეს ელემენტი შეინახავს მიმაგრებული ელემენტის, ანუ თავად ელემენტს მისი ატრიბუტებით.

ისინი შეიძლება გამოყენებულ იქნას მხოლოდ თემფლეითში (და არა მაგალითად ტაიპსკრიპტის კოდში). მაგალითი:

**html:**

1. <input #inputElement type="text" />
2. <button (click)="getInput(inputElement)">Get Input</button>

**ts:**

1. getInput(inputComesHere: HTMLInputElement){
2. console.log(inputComesHere.value)
3. }

**HTMLInputElement**უნდა დავუდეკლარიროთ არგუმენტს ტიპის სახით, რათა უპრობლემოდ მივწვდეთ მას ტაპსკრიპტში **(მასში ინახება თავად value).**

**5. Components & Databinding Deep Dive**

75. Getting Access to the Template & DOM with @ViewChild

**View Child:**

**@ViewChild** დეკორატორი გამოიყენება ასევე **DOM**-ის ელემენტებთან წვდომისთვის. იგი იყენებს ლოქალ რეფერენსს, რათა ელემენტებს მიწვდეს. იგი იმპორტირდება **@angular/core**-დან.

თუ ngOnInit-დან გვსურს ელემენტებს მივწვდეთ, მაშინ:

{ static: true }, სხვა შემთხვევაში: { static: false}

მაგალითი:

**html:**

1. <input type='text' #localRefName/>
2. <h2>{{ inputEl }} </h2>

**ts:**

1. @ViewChild('localRefName', {static: true}) elementSavedHere: ElementRef
2. inputEl;
3. checkInput(){
4. this.inputEl = this.elementSavedHere.nativeElement.value
5. }

**ElementRef**უნდა დაიმპორტდეს **@angular/core**-დან. **ViewChild**ლოქალ რეფერენსისგან განსხვავებით წარმოადგენს ElementRef-ის ტიპს, რომელსაც გააჩნია **nativeElement**ფროფერთი. *იგი აღარ საჭიროებს****HTMLInputElement****-ის ტიპის გაწერას.*

**5. Components & Databinding Deep Dive**

76. Projecting Content into Components with ng-content

**ng-content:**

იგი გვეხმარება კომპონენტისთვის ინფორმაციის გარედან ჩაწოდებაში, ანუ სადაც თავად კომპონენტი გამოიძეხება. მაგალითად:

**მშობელი html:**

1. <app-server>
2. <p>internal</p>
3. </app-server>

**შვილობილი html:**

1. <div>
2. <ng-content></ng-content>
3. </div>

*ng-content-ის გარეშე შუაში მოთავსებული ინფორმაცია იკარგება.*

*იგი საუკეთესო საშუალებაა ფროფერთი ბაინდინგებისგან თავის ასარიდებლად.*

**5. Components & Databinding Deep Dive**

77. Understanding the Component Lifecycle

**LifeCycle:**

როცა ახალი კომპონენტი იქმნება, ანგულარი გადის გადის რამდენიმე სტადიას, ანუ ციკლს, ფაზებს:

* **ngOnChanges**- იგი გამოიძახება მაშინ, როდესაც ახალი კომპონენტი შეიქმნება და შემდეგ უკვე მაშინ, როდესაც **@Input** ფროფერთიები მიიღებენ ახალ მნიშნველობას, ანუ შეიცვლებიან.
* **ngOnInit**- იგი გამოიძახება, როცა კომპონენტი ინიციალიზდება, ანუ ობიექტი შეიქმნება. იგი გამოიძახება **constructor()** მეთოდის შემდეგ.
* **ngDoCheck**- ეს მეთოდი ბევრჯერ გამოიძახება, ანუ ყოველთვის, როცა ცვლილების დეტექტორი გააქტიურდება. ცვლილებების დეტექტორი წარმოადგენს ანგულარის სისტემას, რომელიც ამოწმებს რომელიმე ფროფერთ შეიცვლა თუ არა, ანუ ყოვეელი ასოს ცვლილებაზე, რომელიც თემფლეითში გამოჩნდება, ეს მეთოდი გამოიძახება. იგი ასევე გამოიძახება თუნდაც ისეთი ღილაკის დაჭერისას, რომელიც არაფერს არ ცვლის **DOM**-ში, რადგან მან უნდა შეამოწმოს, რამდე შეიცვალა თუ არა. ეს შეიძლება გამოიყენო მაგალითად ისეთი ცვლილების დეტექციისთვის, რომელსაც სხვაგვარად ვერ შეამოწმებდი მოხდა თუ არა.

**5. Components & Databinding Deep Dive**

77. Understanding the Component Lifecycle

**LifeCycle:**

* **ngAfterContentInit**- იგი გამოიძახება მაშინ, როცა კონტენტი (ng-content) ინიციალიზდება. ანუ ვიზუალურად ჩანს: ის კონტენტი, რომელიც მშობელი კომპონენტის წყალობით **ng-component**-ის გავლით დაემატება ჩვენს კომპონენტს (იგივე ინიციალიზება).
* **ngAfterContentChecked**- მას შემდეგ გამოიძახება, რაც ზემოთ ნახსენები კომპონენტი ვიზუალიზდება და აღნიშნული ვიზუალიზაცია შემოწმდება (იგივე ინიციალიზაცია, ანუ კოდი წაიკითხა).
* **ngAfterViewInit**- იგი გამოიძახება მას შემდეგ, რაც კომპონენტი საბოლოოდ ვიზუალიზდება, ანუ ფიზიკურად გამოვა. (და შვილობილი ელემენტებიც).
* **ngAfterViewChecked**- მას შემდეგ გამოიძახება, რაც ყველა ცვლილება, რაც უნდა მომხდარიყო, ვიზუალურად, ანუ ფიზიკურად აისახა უკვე ან არ ასახულა არაფერი. (და შვილობილი ელემენტებიც).
* **ngOnDestroy**- გამოიძახება, როცა ობიექტის განადგურების წინ (ანუ როცა განადგურების, წაშლის ფაზაშია ელემენტი).

**5. Components & Databinding Deep Dive**

78. Seeing Lifecycle Hooks in Action

**LifeCycle in Action:**

**ngOnChanges**ერთდერთი ციკლური მეთოდია, რომელიც **SimpleChanges**იღებს როგორც არგუმენტის ტიპს, იგი უნდა დაიმპორტდეს @angular/core-დან:

1. ngOnChanges(changes: SimpleChanges ){
2. console.log(changes)
3. }

**SimpleChanges**დააბრუნებს ობიექტს, რომელსაც არგუმენტად ექნება ის ფროფერთი, რომელიც ლოგიკაში დეკლარირებულ ფროფერთიზეა დამოკიდებული, იგი მოიცავს ზოგად ინფორმაციას ამ ფროფერთიზე, ასევე ინფორმაციას იმაზე, თუ პირველად მოხდა ცვლილება თუ არა და რა იყო წინა სტეიტი.

**5. Components & Databinding Deep Dive**

79. Lifecycle Hooks and Template Access

**Template access & Lifecycle hooks:**

**ViewChild**-ით ელემენტს ვერ მივწვდები მაგალითად **ngOnInit**-იდან, რადგან იგი ვიზუალურად ციკლის მაგ ფაზაში არ არის ასახული. ელემენტთან მსგავსი წვდომისთვის საჭიროა **afterViewInit**ციკლის ფაზა.

**5. Components & Databinding Deep Dive**

81. Getting Access to ng-content with @ContentChild

**ContentChild**:

იგი გამოიყენება კონენტის ინიციალიზების ციკლის ფაზაში:

@ContentChild('reference', {static: true}) element: ElementRef;

ასევე იმპორტირდება**@angular/core**-დან და იგივენაირად იმართება, როგორც **ViewChild**-ი.

**Directives Intro:**

არსებობს **ატრიბუტული**და **სტრუქტურული**დირექტივები,

**ატრიბუტულებია**: ისინი ნორმალურ html ატრიბუტებს ჰგავნან, შესაძლოა აკეთებდნენ დათაბაინდინგს ან ივენთ ბაინდინგს.

**სტრუქტურულებია**: ისინიც ნორმალურ html ატრიბუტებს ჰგავნან, თუმცა წინ **\***-ნიშანი აქვთ. გავლენა აქვთ მთლიან დომზე ელემენტის გარშემო (ელემენტების ჩამატება ან წაშლა)

**7. Directives Deep Dive**

91. ngFor and ngIf Recap

**Directives:**

ერთ ელემენტზე ერთზე მეტი სტრუქტურული დირექტივის მიმაგრება შეუძლებელია.

**7. Directives Deep Dive**

93. Creating a Basic Attribute Directive

**Directives Create (attribute):**

ჩვენი ატრიბუტული დირექტივის შესაქმნელად, ცალკე ts ფაილი უნდა შევქმნათ, რომლის სახელში ასევე directive-საც გავითვალისწინებთ. შემდეგ კი კომპონენტის მსგავსად, აქაც უნდა დადეკლარირდეს დეკორატორი:

1. @Directive({ selector: '[appBasicHighlight]' })
2. export class BasicHighlightDirective {}

იმისათვის, რომ ის ელემენტი დავაინჯექტოთ, რომელზეც დირექტივი ზის, უნდა მოვიქცეთ შემდეგნაირად:

1. export class BasicHighlightDirective implements OnInit {
2. constructor(private elementRef: ElementRef) {}
3. ngOnInit() {
4. this.elementRef.nativeElement.style.backgroundColor: 'green'
5. }
6. }

**ElementRef**უნდა დაიმპორტდეს. დირექტივებს ასევე აქვთ **ngOnInit**, რომელშიც რეკომენდირებულია დირექტივის დეკლარირება. დირექტივი უნდა დაიმპორტდეს აპის მოდულში: **declarations**.

**html:**

  <h1 appBasicHighlight>Hello</h1>

\*თუმცა ამ შემთხვევაში DOM-ს პირდაპირ ვწვდებით, რაც არაა რეკომენდირებული

**7. Directives Deep Dive**

94. Using the Renderer to build a Better Attribute Directive

**Directives Create (attribute) advanced:**

**რეკომენდირებულია**შემდეგნაირი წვდომა **DOM**-ზე დირექტივის შექმნისას:

საწყისში ყველაფერი იგივეა და კლასში კი:

1. export class BetterHighglightDirective implements OnInit {
2. constructor( private renderer: Renderer2, private elementRef: ElementRef) {}
4. ngOnInit(){
5. this.renderer.setStyle( this.elementRef.nativeElement, 'background-color', 'blue', [flag ემატება])
6. }
7. }

https://angular.io/api/core/Renderer2 - სხვა მეთოდფები setStyle-ს მსგავსად

**7. Directives Deep Dive**

96. Using HostListener to Listen to Host Events

**Directives Create (attribute) advanced (listen Hosts):**

იმისათვის, თუ გვსურს, რომ დირექტივი ინტერაქტიული იყოს, ანუ ივენთს უსმენდეს, უნდა შემოვიტანოთ **@HostListener @angular/core-**დან:

1. export class BetterHighlightDirective {
2. constructor( private renderer: Renderer2, private elementRef: ElementRef) {}
3. @HostListener('mouseenter') mouseenters(eventData: Event) {
4. this.renderer.setStyle( this.elementRef.nativeElement, 'background-color', 'blue', [flag ემატება])
5. }
6. @HostListener('mouseleave') mouseleaves(eventData: Event) {
7. this.renderer.setStyle( this.elementRef.nativeElement, 'background-color', 'transparent', [flag ემატება])
8. }
9. }

**7. Directives Deep Dive**

97. Using HostBinding to Bind to Host Properties

**Directives Create (attribute) advanced (no renderer):**

Renderer2-ის ნაცვლად შეგვიძლია გამოვიყენოთ **@HostBinding**, რომელიც ასევე უნდა დავაიმპორტოთ **@angular/core**-დან. **HostBinding**-ის არგუმენტში მიდის ის **ატრიბუტი**, რომელსაც უნდა მივწვდეთ.

1. export class BetterHighlightDirective {
2. @HostBinding('style.backgroundColor') backgroundColor: string = 'transparent';
3. constructor() {}
4. @HostListener('mouseenter') mouseenters(eventData: Event) {
5. this.backgroundColor = 'blue'
6. }
7. @HostListener('mouseleave') mouseleaves(eventData: Event) {
8. this.backgroundColor = 'transparent'
9. }
10. }

**7. Directives Deep Dive**

98. Binding to Directive Properties

**Costum property binding in Directive:**

ამისათვის უნდა შემოვიტანოთ**@Input** დეკორატორი:

1. export class BetterHighlightDirective implements OnInit {
2. @Input() defaultColor: string = 'transparent';
3. @Input('appHighlightColor') highlightColor: string = 'blue';
4. @HostBinding('style.backgroundColor') backgroundColor:string;
6. ngOnInit(){
7. this.backgroundColor = this.default;
8. }
10. @HostListener('mouseenter') mouseenters(eventData:Event) {
11. this.backgroundColor = this.highlightColor
12. }
13. @HostListener('mouseleave') mouseleaves(eventData:Event) {
14. this.backgroundColor = this.defaultColor
15. }
16. }

**html:**

<p appHighlightColor [defaultColor]="'yellow'" [highlightColor]="'blue'"> Hello Directive </p>

**ასევე შესაძლოა:**

<p defaultColor="yellow" [appHighlightColor ]="'blue'"> Hello Directive </p>

**7. Directives Deep Dive**

99. What Happens behind the Scenes on Structural Directives

**Structural Directives Understanding:**

ანგულარი სტრუქტურული დირექტივის გადაჰყავს სხვა ფორმაში, რომელიც შეძლებს ატრიბუტის გაგებას, ანუ წაშლის **\***- ნიშანს.

ასევე შეგვიძლია გამოვიყენოთ შემდეგნაირად: \*ngIf="true"-ის ნაცვლად [ngIf]="true" **ng-template**-ში

**7. Directives Deep Dive**

100. Building a Structural Directive

**Create Structural Directive:**

დირექტივის შექმნა **CLI**-დან: ng g d [სახელი]

დანარჩენი ყველაფერი იგივე, შემდეგ კი:

1. export class UnlessDirective {
2. @Input() set appUnless(condition: boolean) {
3. if(!condition){
4. this.vsRef.createEmbededView(this.templateRef)
5. } else {
6. this.vcRef.clear();
7. }
9. }
11. constructor(private: templateRef: TemplateRef<any>, private vcRef: ViewContainerRef) { }
12. }

**TemplateRef**და **ViewContainerRef**უნდა დავაიმპორტოთ **@angular/core**-დან.

მნიშნველოვანი: **set**ფროფერთიმ იგივე სახელი უნდა გაიზიაროს, რაც დირექტივს აქვს, სხვაგვარად არ იმუშავებს.

**7. Directives Deep Dive**

101. Understanding ngSwitch

**Directive \*ngSwitch:**

ისიც ქონდიშენალს წარმოადგენს, მაგალითი:

**html:**

1. <div [ngSwitch]="value">
2. <p \*ngSwitchCase="5">Value is 5</p>
3. <p \*ngSwitchCase="10">Value is 10</p>
4. <p \*ngSwitchCase="100">Value is 100</p>
5. <p \*ngSwitchDefault>Value is default</p>
6. </div>

**ts:**

value: number = 10

**9. Using Services & Dependency Injection**

106. Creating a Logging Service

**Services:**

სერვისის შექმნისას აუცილებელია გავითვალისწინოთ სახელში **.service** მონაკვეთი.

სერვისებს არ ჭირდებათ დეკორატორი. ახლა ვაკეთებთ მის ყველაზე პრიმიტიულ ვარიანტს.

**სერვისის ts:**

1. export class LoggingService {
2. logStatusChange(status: string){
3. console.log('სერვისი მუშაობს');
4. }
5. }

**სხვა კომპონენტში იმპორტირდება:**

1. import { LoggingService } from './logging.service'
2. export class Component {
3. doSmth(){
4. const service = new LoggingService();
5. service.logStatusChange('სტატუსი')
6. }
7. }

თუმცა ამის გამოყენება***არაა რეკომენდირებული.***

**9. Using Services & Dependency Injection**

107. Injecting the Logging Service into Components

**Services dependency injection, structure:**

დიფენდენსი ინჯექშენი სერვისის კლასის ინსტანსს აინჯექტებს ჩვენს კომპონენტში ავტომატურად.

**სერვისის ts:**

1. export class LoggingService {
2. logStatusChange(status: string){
3. console.log('სერვისი მუშაობს');
4. }
5. }

**სხვა კომპონენტში ინჯექშენი:**

1. import { LoggingService } from './logging.service'
2. @Component({
3. providers: [LoggingService] //ამით ვატყობინებთ, რომ ეს კონკრეტული სერვისი უნდა გამოვიყენოთ.
4. })
5. export class Component {
6. constructor(private loggingService: LoggingService){ }
7. doSmth(){
8. this.loggingService.logStatusChange('სტატუსი')
9. }
10. }

**provider**-ში დეკლარირებულ სერვისზე წვდომა აქვს როგორც მშბელ კომპონენტს, ასევე შვილობილებს (ანუ **provider**-ში შვილობილში აღარ ვადეკლარირებთ).

**9. Using Services & Dependency Injection**

109. Understanding the Hierarchical Injector

**Service Hierarchy:**

როდესაც მშობლის **priveder**-შია დეკლარირებული სერვისი, მისი შვილობილებში საჭირო აღარაა, რადგან ახალი ინსტანსი იქნება და გადაწერა ხდება და იბაგება.

სერვისების იერარქია, რომ **provider**-ში დეკლარირება:

**1. AppModule**- მთელს აპლიკაციაში იქნება ხელმისაწვდომი სერვისი (აუცილებელია, როცა **სერვისი უნდა დავაინჯექტოთ** სხვა სერვისით)

**2. AppComponent**- ამ მშობელს და მის შვილებილებსაც იგივე სერვისი ექნებათ;

**3. any other Component**- რომელსაც შვილობილი არ ჰყავს.

**9. Using Services & Dependency Injection**

111. Injecting Services into Services

**Service injection into Service:**

იმისათვის, რომ სერვისი დავაინჯექტოთ სხვა სერვისით, ის სერვისი, რომლითაც ვაინჯექტებთ, **AppModule**-ის **providers**-ში უნდა დადეკლარირდეს.

პირველ რიგში კონსტრუქტორი უნდა დავამატოთ, დავაინჯექტოთ სხვა სერვისით (სტანდარტულად), თუმცა კლასის ზევით უნდა დადეკლარირდეს**@Injectable(),** რომელიც უნდა დაიმპორტდეს **@angular/core**-დან, სხვაგვარად არ იმუშავებს.

**მიმღები სერვისის ts:**

1. @Injectable()
2. export class LoggingService {
3. constructor( private otherService: OtherService ) { }
4. logStatusChange(status: string){
5. console.log('სერვისი მუშაობს');
6. }
7. }

**9. Using Services & Dependency Injection**

112. Using Services for Cross-Component Communication

**Service კომპონენტებს შორის ურთიერთობა:**

ივენთის ემიტინგი შეგვიძლია სერვისში, რომელსაც დაატრიგერებს რომელიმე შვილობილი და ამ დატრიგერებულს მოუსმენს სხვა შვილობილი, მაგალითად:

**სერვისის ts:**

statusUpdated = new EventEmitter<string>();

**ერთი კომპონენტი:**

this.service.statusUpdated.emit('Hello')

**მეორე კომპონენტი:**

1. this.service.statusUpdated.subscribe(
2. (status: string) => console.log('სერვისი მიღებულია' + status)
3. )

***\*შენიშვნა:***მშობელ კომპონენტში სერვისის შემდეგი სახით დეკლარირება საჭიროს აღარ ხდის სერვისის AppModule-ში დეკლარირებას და ისეა ხელმისაწვდომი ყველგან: @Injectable({providedIn: 'root'})

**11. Changing Pages with Routing**

126. Setting up and Loading Routes

**Routing introduction 1:**

როუტინგის ინიციალიზება შეიძელება ორგვარად: დამოუკიდებლად ასევე **appModule**-ში. ამ ჯერზე **appModule**-ში ვაკეთებთ.

ამისთვის უნდა შევქმნათ კონსტანტი ცვლადი, რომელიც იქნება Route-ს ტიპის, რომელსაც დავაიმპორტებთ**@angular/router**(სასურველია ზევით).

1. app.module.ts:
2. const appRoute: Routes = [
3. { path: '', component: HomeComponent}, //empty path
4. { path: 'users', component: UsersComponent },
5. { path: 'servers', component: ServersComponent }
6. ]

ამ პათის გამოყენებისთვის, **@NgModule**-ის **imports**-ის მასივში უნდა ჩავსვათ **RouterModule**და იგი უნდა დაიმპორტდეს **@angular/router**-დან. **RouterModule**-ს უნდა მივაბათ **forRoot()** და არგუმენტად ჩავაწოდოთ **appRoutes**(ანუ რაც ცვლადს დავარქვით:

1. import { Routes, RouterModule } from '@angular/router'
2. @NgModule({
3. imports: [
4. RouterModule.forRoot(appRoutes)
5. ]
6. })

**11. Changing Pages with Routing**

126. Setting up and Loading Routes

**Routing introduction 2:**

შემდეგ აღნიშნული როუტინგი თემფლეიტში უნდა დავადეკლარიროთ. კომპონენტები, რომლებზეც რაუტინგი ვრცელდება, მათი სელექტორების გამოძახება აღარაა საჭირო, მხოლოდ <router-outlet></router-outlet> არის საჭრო.

**11. Changing Pages with Routing**

127. Navigating with Router Links

**Routing introduction router linking:**

ნავიგაციის ასამუშავებლად შეგვიძლია **<a>** თაგის **href**-ატრიბუტში ჩავწეროთ **"/routes"** და იმუშავებს, თუმცა არეფრეშებს რა ყოველ ჯერზე თავიდან კითხულობს მთლიან კოდს, რაც არ არის კარგი.

მის ალტერნატივად შეგვიძლია გამოვიყენოთ: **routerLink="routes"**:

1. <a routerLink="/">Home</a>
2. <a routerLink="/servers">Home</a>
3. <a routerLink="/users">Home</a>

ან ალტერნატივად შეგვიძლია ფროფერთი ბაინდინგით:

<a [routerLink]="['/users', 'something' ]"

მას თავისი უპირატესობებიც აქვს. მაგალითად, მასივში ყველა იმ **სეგმეტს**წერ, რომელიც ფათმა უნდა გაიაროს.

**11. Changing Pages with Routing**

128. Understanding Navigation Paths

**router linking:**

თუ **/ (სლეშების)** გარეშე დავადეკლარირებთ გალინკვისას:

<a routerLink="servers"> Servers </a>

საწყისზე არაფერი არ შეიცვლება. თუმცა თუ მაგალითად servers კომპონენტშივე დავადეკლარირებდით იგივე როუტის ლინკს, რომელზე დაჭერისასაც იგივე ლოკაციაზე მიდის სადაც თავად არის, დაიბაგებოდა! ამის თავიდან ასაცილებლად, **/ (სლეში)** უნდა ჩავუმატოთ თავად servers-ის კომპონენტში, რათა იგი absolute **path**-ად ვაქციოთ.

**relative path: "servers" (ან "./servers") -**იგი ამაგრებს მითითებულ მისამართს მიმდინარე მისამართს, ანუ **https://localhost:4200/servers/servers .***იგი შეიძლება სასარგებლო იყოს ჩანესტებული ბილიკების შემთხვევაში.*

**absolute path: "/servers"  (ან "../servers") -**ამ შემთხვევაში მიმდინარე მისამართზე პირდაპირ არ ებმება ახალი მისამართი, იგი უბრალოდ ნაცვლდება **https://localhost:4200/servers .**

**11. Changing Pages with Routing**

129. Styling Active Router Links

**Active Routes 1:**

აღნიშნული **css**-ის საქმეა, ანუ ჯერ **css**-ში უნდა შეიქმნას შესაბამისი სტილი და ამ სტილის დინამიურად მიმაგრებისთვის უკვე, ანგულარ იგვაძლევს შემდეგ ატრიბუტულ დირექტივს: **routerLinkActive=""**, სადაც უნდა ჩაჯდეს კლასი. აღნიშნული დირექტივეი ამოწმებს რომელ **path**-ში ხარ და ადარებს დაკლიკებულ ლინკს, თუ ემთხვევიან, იგი აბამს მითითებულ კლასს კონკრეტულ ლინკს.

1. <a routerLinkActive="active" routerLink="/">Home</a>
2. <a routerLinkActive="active" routerLink="/servers">Servers</a>
3. <a routerLinkActive="active" routerLink="/users">Users</a>

თუმცა წარმოიქმნება ერთი პრობლემა: ის **ლინკი**, რომელსაც ცარიელი **path**აქვს, ყოველთვის ექნება გააქტიურების კლასი მიბმული, რადგან ცარიელი **path**ყველა **path**-ის ნაწილია. ამის თავიდან ასარიდებლად გვაქვს დირექტივი, რომელიც ამ ცარიელ **path**-იან ლიკის თაგს უნდა დაემატოს:

[routerLinkActiveOptions]="{exact: true}"

**11. Changing Pages with Routing**

129. Styling Active Router Links

**Active Routes 2:**

[routerLinkActiveOptions]="{exact: true}"

იგი ეუბნება ანგულარს, რომ აღნიშნულ თაგს მიაბი აქტიური კლასი მხოლოდ მაშინ, თუ მითითებული **path**ზუსტად ემთხვევა, ანუ სრულიად ცარიელია.

**11. Changing Pages with Routing**

130. Navigating Programmatically

**Routing logic (ts) 1:**

როუტინგის ტრიგერინგი ასევე შეიძლება ლოგიკის **(ts)** ფაილიდან.

მაგალითად, **html**-დან მივწვდეთ ამ ლოგიკას:

<button (click)="onLoadServers()" >Load Server</button>

**ts:**

1. constructor(private router: Router){}
2. onLoadServers(){
3. this.router.navigate(['/servers']) //absolute path
4. }

Router უნდა დაიმპორტდეს, როგორც წინაზე. ეს როუტერი იღებს მეთოდს **navigate([]),** რომელშიც **path**-ის მასივი მიდის.

**11. Changing Pages with Routing**

131. Using Relative Paths in Programmatic Navigation

**Routing logic (ts) 2:**

ლოგიკაში რელატიური **path**-იც შეგვიძლია გამოვიყენოთ. თუ მას პირდაპირ დავადეკლარირებთ ( this.router.navigate(['servers']) ), იგი ერორს არ მოგვცემს. რადგან **navigate** მეთოდმა **routerLink**-სგან განსხვავებით, არ იცის იმ მომენტისთვის რომელ როუტში (**path**-ზე) იმყოფები.

თუმცა შეგვიძლია მას მივაწოდოთ ინფორმაცია, თუ **სად ვიმყოფებით:**

1. constructor(private router: Router, private route: ActivatedRoute){}
2. onLoadServers(){
3. this.router.navigate(['servers'], {relativeTo: this.route}) //relative path
4. }

**ActivatedRoute**უნდა დაიმპორტდეს **@angular/router**-დან

**11. Changing Pages with Routing**

132. Passing Parameters to Routes

**Route parameters 1:**

თუ გვსურს ინდივიდუალური კომპონენტი, მაგალითად **users**სათითაოდ ჩავაწოდოთ როუტს, ანუ რომ აჩვენოს რომელი **user**-ა გახსნილი, ერთი ვარიანტია:

1. app.module.ts:
2. const appRoute: Routes = [
3. { path: '', component: HomeComponent}, //empty path
4. { path: 'users', component: UsersComponent },
5. { path: 'users/1', component: UserComponent },
6. { path: 'users/2', component: UserComponent },
7. { path: 'users/3', component: UserComponent },
8. { path: 'servers', component: ServersComponent }
9. ]

ანუ სათითაოდ დავადეკლარიროთ **path**-ში.

ამის ნაცვლად შეგვიძლია დინამიურად შევიტანოთ პარამეტრები :id-ის გამოყენებით, მაგ **'users/:id'**. **:**-ს ეუბნება ანგულარს, რომ პარამეტრი დინამიური იქნება.

{ path: 'users/:id', component: UserComponent }

აღნიშნული დინამიური path საშუალებას მოგვცემს, რომ

**https://localhost:4200/user/რაც არ უნდა ჩაწერო -**გადაგვიყვანს UserComponent-ზე. მაგალითად:

https://localhost:4200/user/something

**11. Changing Pages with Routing**

133. Fetching Route Parameters

**Route parameters 2:**

ახლა ჩვენ უნდა მივწვდეთ **path**-ს, ანუ ლინკის პარამეტრს. ამისთვის ლოგიკაში **(ts)** (**currently loaded route** წარმოადგენს ჯავასკრიპტის ობიექტს, რომელიც დიდი რაოდენობით მეტა-ინფორმაციას შეიცავს, როგორც ყველა ელემენტი, რომელიც ობიექტს წარმოადგენს):

1. user: {id: number, name: string};
2. constructor(private route: ActivatedRoute){}
3. ngOnInit(){
4. this.user = {
5. id: this.route.snapshot.params['id'],
6. name: this.route.snapshot.params['name']
7. }
8. }

**appModule**-ში ასევე ჩამატებული გვაქვს ახალი დინამიური მისამართი:

{ path: 'users/:id/**:name**', component: UserComponent }

ამ ინფორმაციით კომპონენტის თემფლეითში **(html):**

1. <p>Name: {{ user.name }} </p>
2. <p>ID:  {{ user.id}} </p>
3. **11. Changing Pages with Routing**
4. 134. Fetching Route Parameters Reactively
5. **Route parameters 3.1:**
6. წინაზე მოცემული მაგალითი ყველა შემთხვევაში შეიძლება არ მუშაობდეს, მაგალითად თუ როუტს გვსურს რომ ღილაკზე **დაკლიკებით**მივწვდეთ:
7. <a [routerLink]="['/users',10,'anna']">Load User</a>
8. აღნიშნული დაააფდეითებს ლინკს, თუმცა ინფორმაცია ობიექტს არ ჩაეწოდება, რადგან როდესაც ანქორ თაგი დეკლარირებულია იმ კომპონენტშვე, რომლის შიგა როუტში გადასვლაც გვსურს, ანგულარი მთლიან კომპონენტს არ დაარენდერებს, შესაბამისად **ngOnInit არ იმუშავებს**, მხოლოდ ლინკს დაააფდეითებს.
9. ამის გამოსასწორებლად **ActivatedRoute**-ს გააჩნია **params**პარამეტრი, იგი წარმოადგენს **Observable**-ს (იგი საშუალებას გვაძლევს ასინქრონულ დავალებებზე ვიმუშაოთ). იგი მიიბამს მეთოდს - **subscribe():**

**11. Changing Pages with Routing**

134. Fetching Route Parameters Reactively

**Route parameters 3.2:**

1. constructor(private route: ActivatedRoute){}
3. ngOnInit(){
4. this.user = {
5. id: this.route.snapshot.params['id'],
6. name: this.route.snapshot.params['name']
7. }
8. this.route.params.subscribe(
9. ( params: Params ) => {
10. this.user.id = params['id'],
11. this.user.name = params['name']
12. }
13. );
14. }

**subscribe**იღებს **3 არგუმენტს**, თუმცა პირველია ყველაზე მნიშნველოვანი. **Params**-იც ობიექტს წარმოადგენს და უნდა დაიმპორტდეს **@angular/router**-დან. იგი დააფდეითდება მას მერე, რაც პარამეტრები შეიცვლება ავტომატურად *(ngOnInit-ის მიუხედავად)*.

***ეს არდაგვჭირდება იმ შემთხვევაში, თუ ვიცით, რომ კომპონენტი არ დალოადდება შიგნითვე.***

**11. Changing Pages with Routing**

135. An Important Note about Route Observables

**Route parameters 3.3:**

კიდევ ერთი ფაქტია: როდესაც **subscribe**-ს ვაკეთებთ, აღნიშნული **path** ინახება მემორიში კომპონენტიდან გასვლის, ანუ განადგურების შემთხვევაშიც კი. აღნიშნული გაკონტროლება **onDestroy life-cycle**-ით შეიძლება:

1. paramsSubscription: Subscription;
2. ngOnDestroy(){
3. this.paramsSubscription.unsubscribe()
4. }

**Subscription**უნდა დაიმპორტდეს**rxjs/Subscription**-დან, რადგან იგი წარმოადგენს **observable**-ს, რაც გარე პაკეტია, რომელსაც ანგულარი იყენებს.

*\*ამის გაკეთება აუცილებელი არაა, რადგან ანგულარი დამოუკიდებლად იზამს. მხოლოდ მაშინაა საჭირო, თუ საკუთარი observable შექმენი. თუმცა არც ამის გაკეთება არ ავნებს app-ს.*

**11. Changing Pages with Routing**

136. Passing Query Parameters and Fragments

**Routing Query Params and Fragments 1:**

**Query Parameters** - ისინი ლინკში კითხვისნიშნით არიან გამოყოფილნი - *https://localhost:4200/home/person?something=smth*

**Fragments**- იგივე ჰაშ ფრაგმენტი (#). ის შეიძლება მხოლოდ ერთი გქონდეს -  *https://localhost:4200/home/person?something=smth#loading*

***ინფორმაციის ჩაწოდება ქიური პარამეტრში და ფრაგმენტში:***

მაგალითზე ჩავამატოთ ახალი როუტი **appModule**-ში:

{ path: 'servers/:id/edit', component: EditServerComponent }

შემდეგ **html**-ში ჩავამატოთ **routerLink**:

1. <a
2. [routerLink]="['/servers',5,'edit']
4. [queryParams]="{ allowEdit: '1' }" //უფრო მეტი შეგვიძლაი გვქონდეს მნიშვნელობები, რომლებიც &-ნიშნით იქნებოდნენ გამოყოფილები
6. fragment="loading"
8. >Edit File</a>

**queryParams**არ წარმოადგენს ახალ დირექტივს, იგი **routerLink**-ზე მიბმადი ფროფერთია.

**11. Changing Pages with Routing**

136. Passing Query Parameters and Fragments

**Routing Query Params and Fragments 2:**

აღნიშნულის ლოგიკურად დამატებაც შეიძლება (ts) ფაილში

ჯერ **html**-ში ღილაკი შევქმნათ:

<button (click)="onLoadServer(1)">Load server 1</button>

შემდეგ კი **ts**-ში:

1. constructor( private router: Router) {}
2. onLoadServer(id: number){
3. this.router.navigate( ['/servers',id, 'edit'], {queryParams: { allowEdit: '1' }, fragment: 'loading'} )
4. }

*აქ ვწვდებით ლინკს მასზე გადასვლით...*

**11. Changing Pages with Routing**

137. Retrieving Query Parameters and Fragments

**Routing Query Params and Fragments 3:**

ამ კონკრეტული პარამეტრების მიღებისთვის ისევ **ActivatedRoute**-ს ვაინჯექტებთ კონსტრუქტორში.***წინას მსგავსად, 2 გზაა***ამ მონაცემების **მისაღებად**:

1. query;
2. fragment;
3. constructor( private route: ActivatedRoute ){}
4. ngOnInit(){
5. console.log(this.route.snapshot.queryParams)
6. console.log(this.fragment = this.route.snapshot.fragment)
7. }

**მეორე:**

1. query;
2. fragment;
3. constructor( private route: ActivatedRoute ){}
4. ngOnInit(){
5. this.route.queryParams.subscribe()
6. this.route.fragment.subscribe()
7. }

**11. Changing Pages with Routing**

139. Setting up Child (Nested) Routes

**Routing nested child routes:**

ამისთვის **appModule**-ში არსებულ როუტინგის მასივში მშობელ როუტ ობიექტს კომპონენტის შემდგომ უნდა ჩავამატოთ **children**- ფროფერთი და იგი უნდა წარმოადგენდეს ახალ მასივს, რომელშიც ჩავყრით ჩაილდ როუტებს, მათ კი **path**-ებში მშობლის მისამართს მოვაშორებთ *(თუ აქვთ)*:

**app.module.ts:**

1. const appRoute: Routes = [
2. { path: '', component: HomeComponent}, //empty path
3. { path: 'users', component: UsersComponent, children: [
4. { path: ':id/:name', component: UserComponent },
5. ] },
6. { path: 'servers', component: ServersComponent, children: [
7. { path: ':id', component: ServerComponent }
8. { path: ':id/edit', component: EditServerComponent }
9. ]
10. }
11. ]

შემდეგ კი რომელიმე *(ამ შემთხვევაში იუზერის და სერვერის კომპონენტში)* უკვე დაროუტებულ ელემენტში კიდევ ერთი როუტინგი უნდა ჩავამატოთ, ანუ: <router-outlet></router-outlet>

**11. Changing Pages with Routing**

141. Configuring the Handling of Query Parameters

**Routing queryParamsHandling:**

აღნიშნული იწერება როუტერის ნავიგაციის მოდულის მეორე არგუმენტის ობიექტში.

1. this.router.navigate(['edit'], {relativeTo: this.route, queryParamsHandling: 'preserve'})

**queryParamsHandling-ს შეუძლია მიიღოს 2 მნიშნელობა:**

**"marge"** - ძველი პარამეტრების ახალთან გაერთიანებისთვის

**"preserve"** - ინარჩუნებს ძველ პარამეტრებს და ახლებს დროპირებას უკეთებს.

**11. Changing Pages with Routing**

142. Redirecting and Wildcard Routes

**Routing გადამისამართება:**

სასურველია მისთვის **ცალკე კომპონენტი** გვქონდეს, რომელზეც გადამისამართდება როუტინგი. რედაირექტი/გადამისამართება ნებისმიერ როუტინგის ობიექტს შეგვიძლია ჩავამატოთ: **redirectTo: '/სადაც წავიდეს' .**

{ path: 'servers', redirectTo: PageNotFoundComponent }

ამ შემთხვევაში ჩვენ გვსურს გადავამისამართოთ ამ კონკრეტულ კომპონენტში ყველა ხელით შეყვანილი path, რომელიც როუტინგში არ არსებობს, ამისთვის ვიყენებთ **wildCardRoute**-ს **path**-ში **\*\***-ის დამატებით:

1. { path: 'not-found', component: PageNotFoundComponent }
2. { path: '\*\*', redirectTo: '/not-found' }

***\*\* - იგი ბოლოში უნდა იყოს აუცილებლად***

იმისათვის, რომ გადამისამართების ერორი ავიცილოთ თავიდან, იმ შემთხვევაში, თუ ჰომიდან ხდება გადამისამართება, ანუ სუფთა ლინკიდან, **pathMatch: 'full'**უნდა ჩავამატოთ გადამისამართების როუტინგის ობიექტს:

{ path: '', redirectTo: '/somewhere-else', pathMatch: 'full' }

**11. Changing Pages with Routing**

144. Outsourcing the Route Configuration

**Routing მოდულის დამოუკიდებლად შექმნა (appModule-ის გარეთ):**

ტიპიურად, აღნიშნულ ფაილს ქმნიან შემდეგი სახელით:**app-routing.module.ts**.

მისი **სხეული**შემდეგნაირად გამოიყურება:

1. import { NgModule } from '@angular/core'
2. import {ყველაფერი რაც საჭიროა}
3. აქ დეკლარირდება const appRoutes: Routes = [მთელი ინფორმაცია როუტზე]
5. @NgModule({
6. imports: [
7. RouterModule.forRoot(appRoutes)
8. ],
9. exports: [RouterModule]
10. })
11. export class AppRoutingModule {}

**appModule**-დან უნდა წავშალოთ **RouterModule.forRoot().**

***exports****ეუბნება ანგულარს, რისი გატანა გვსურს ჩვენი მოდულიდან სხვა მოდულში*

**app.module**-ში კი ვაიმპორტებთ **AppRoutingModule**-ს:

1. @NgModule({
2. imports: [
3. AppRoutingModule
4. ],
5. })

**11. Changing Pages with Routing**

146. Protecting Routes with canActivate

**Routing Guard (canActivate) 1.1:**

აღნიშნულს ანგულარი კითხულობს კონკრეტული კომპონენტის წაკითხვამდე და შესაბამისად იღებს გადაწყვეტილებას, წაიკითხოს თუ არა კომპონენტი. ამისთვის სასურველია შევქმნათ სერვისი,მაგ: **auth-guard.service.ts:**

1. import { CanActivate, ActivatedRouterSnapshot, RouterStateSnapshot } from '@angular.router'
3. import { Observable } from 'rxjs/Observable';
4. export class AuthGuard emplements CanActivate {
5. canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
7. }
8. }

ასევე შევქმნათ სხვა სევისი, რომელიც შეამოწმებს აუთენთიფიკაციის სტეიტს **auth.service.ts:**

1. export class AuthService {
2. loggedIn = false;
4. isAuthenticated() {
5. const promise = new Promise (
6. (resolve, reject) => {
7. setTimeout( () => { resolve(this.loggedIn) }, 8000)
8. }
9. )
10. return promise
11. }
12. login() {
13. this.loggedIn = true;
14. }
15. logOut(){
16. this.loggedIn = false
17. }
18. }

**11. Changing Pages with Routing**

146. Protecting Routes with canActivate

**Routing Guard (canActivate) 1.2:**

წინაში მოცემული ორივე სერვისი **@Injectable()**-თი უკავშირდებიან ერთმანეთს და ამოწმებენ, მომხმარებელი დალოგინებულია თუ არა (***true or false***).

auth-guard.service.ts-ში აღნიშნული შემდეგნაირად მოწმდება:

1. export class AuthGuard implements CanActivate {
2. constructor(private authService: AuthService, private router: Router) {}
3. canActivate(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
4. return this.authService.isAuthenticated().then(
5. (authenticated: boolean) => {
6. if(authenticated){
7. return true }
8. else{
9. this.router.navigate(['/']);
10. }
11. }
12. )
13. }
14. }

ამის შემდეგ კი მივდივართ როუტინგ მოდულში და ვირჩევთ იმ როუტინგის ობიექტს, რომელზეც გვინდა ეს გავრცელდეს (იგი ასევე შეიძლება გავრცელდეს შვილობილ როუტინგებზეც) და:

{ path: 'servers', component: ServersComponent, canActivate: [AuthGuard] }

**11. Changing Pages with Routing**

146. Protecting Routes with canActivate

**Routing Guard (canActivate) 1.3:**

შექმნილი სერვისებიც ასევე უნდა დაიმპორტდნენ **appModule**-ის **provider**-ებში, რათა გლობალურად იყვნენ ხელმისაწვდომნი.

**11. Changing Pages with Routing**

147. Protecting Child (Nested) Routes with canActivateChild

**Routing Guard (canActivateChild):**

**canActivateChild**შეგვიძლია გამოვიყენოთ იმ შემთხვევაში, თუ გვსურს რომ გარდი ყველა შვილობილ როუტზე გავრცელდეს;

**auth-guard.service.ts-**ს ჩაემატება ჯერ იმპლემენტაციაში - **CanActivateChild**და შემდგომ როგორც მეთოდი შიგნით:

1. export class AuthGuard emplements CanActivate, CanActivateChild {
2. canActivateChild(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
3. return this.canActivate(route, state)
4. }
5. }

**როუტინგის**მოდულში კი:

{ path: 'servers', component: ServersComponent, canActivateChild: [AuthGuard] }

**11. Changing Pages with Routing**

149. Controlling Navigation with canDeactivate

**Routing Guard (canDeactivate) 1.1:**

**canDeactivate-**ით შეგვიძლია გავაკონტროლოთ მომხმარებელს შეუძლია თუ არა გასვლა, ანდაც ზოგადად სადმეს კომპონენტიდან გადასვლა, ანუ კომპონენტის დატოვება. სასურველია გვქონდეს ამისთვის ცალკე სერვისი, ამ შემთხვევაში სახელად **can-deactivate-guard.service.ts**, შიგნით კი:

1. import { Observable } from 'rxjs/Observable';
2. import { CanDeactivate } from '@angular.router'
3. export interface CanComponentDeactivate{
4. canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
5. }
6. export class CanDeactivateGuard implements CanDeactivate<CanComponentDeactivate> {
7. canDeactivate(component: CanComponentDeactivate, currentRoute: ActivatedRouteSnapshot, currentState: RouterStateSnapshot, nextState?: RouterStateSnapshot): Observable<boolean> | Promise<boolean> | boolean {
8. return component.canDeactivate();
9. }
10. }

შემდეგ როუტინგის მოდულში მივდივართ და სასურველ როუტინგის ობიექტს ვაწვდით ახალ ფროფერთის: canDeactivate: [CanDeactivateGuard].

**11. Changing Pages with Routing**

149. Controlling Navigation with canDeactivate

**Routing Guard (canDeactivate) 1.2:**

აუცილებელია **CanDeactivateGuard**-ის დაიმპორტება და აღნიშნული სერვისის დეკლარირება **providereb**-ში როუტინგის მოდულში.

იმისათვის რომ ეს ავამუშავოთ, ასევე იმ კომპონენტში უნდა წავიდეთ, რომელსაც როუტინგში აღნიშნული გავუწერეთ და მისი კლასის **implements**-ში უნდა გავუწეროთ ჩვენი ინტერფეისი **CanComponentDeactivate**(დაიმპორტდება შესაბამისად).

შემდეგ კი **შიგნით კომპონენტში**:

1. canDeactivate():Observable<boolean> | Promise<boolean> | boolean; {
2. if(!this.allowEdit) {
3. return true
4. }
5. if( (this.serverName !== this.server.name || this.serverStatus !== this.server.status) && !this.changesSaved) {
6. return confirm('Do you want to discard the changes?')
7. } else {
8. return true
9. }
10. }

**11. Changing Pages with Routing**

150. Passing Static Data to a Route

**Routing (passing static data) სტატიკური ინფორმაციის ჩაწოდება:**

მის გამოსაყენებლად როუტინგ მოდულში ერორის როუტის ობიექტს უნდა ჩავაწოდოთ სტატიკური დათა ფროფერთიდ:

1. {path: 'not-found', component: ErrorPageComponent, data: {message: 'Page not found!'}}

შემდეგ კი კონკრეტული **ერორის კომპონენტში:**

1. constructor(private route: ActivatedRoute) { }
3. ngOnInit(): void {
4. this.errorMessage = this.route.snapshot.data['message']
6. this.route.data.subscribe(
7. (data: Data) => {
8. this.errorMessage = data['message']
9. }
10. )
11. }

**11. Changing Pages with Routing**

151. Resolving Dynamic Data with the resolve Guard

**Routing (passing dynamic data) დინამიური ინფორმაციის ჩაწოდება 1.1:**

ამ შემთხვევაში ვიყოვებთ რესოლვერს (**Resolver**), რომელიც კომპონენტის დარენდერებამდე დარენდერდება, თუმცა იგი კომპონენტის დარენდერებას ხელს არ შეუშლის, ანუ მაინც დაარენდერებს, თუმცა მას მიაბამს ახალ ინფორმაციას.

იგი განსაკუთრებით გამოიყენება ასინქრონულ მონაცემებთან.

მისთვის ცალკე დეკლარირდება სერვისი შემდეგი სახით:

1. import { Observable } from "rxjs";
2. interface Server {
3. id: number;
4. name: string;
5. status: string
6. }
7. @Injectable()
9. export class ServerResolver implements Resolve<Server>{
10. constructor(private serversService: ServersService){}
12. resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<Server> | Promise<Server> | Server{
13. return this.serversService.getServer(+route.params['id'])
14. }
15. }

**11. Changing Pages with Routing**

151. Resolving Dynamic Data with the resolve Guard

**Routing (passing dynamic data) დინამიური ინფორმაციის ჩაწოდება 1.2:**

შემდეგ იმ კონკრეტულ კომპონენტთან მივდივართ, რომელსაც ვწვდებით:

1. this.route.data.subscribe(
2. (data: Data) => {
3. this.server = data['server'];
4. }
5. );

**this.server = data['server']**- აქ '**server**'-ში რა ეწერება დამოკიდებულია იმაზე, თუ როუტინგის მოდულში კონკრეტულ ობიექტს **resolve**-ფროფერთის მნიშნველობად რას აბრუნებინებ.

როუტინგ მოდულში, სამიზნე როუტინგ ობიექტს ვაბამთ შემდეგ ფროფერთის:

1. {path: ':id', component: ServerComponent, resolve: {server: ServerResolver} }

და რესოლვერს ასევე ვაიმპორტებთ **appModule**-ში.

**11. Changing Pages with Routing**

152. Understanding Location Strategies

**Routing url strategies:**

როდესაც სერვერზე ვჰოსტავთ საიტს, მის როუტინგს პირველ რიგში აპარსინგებს სერვერი და შემდეგ მაგალითად ანგულარი, ამიტომ, ანგულარის როუტინგი დიდი შანსია, რომ სერვერისთვის არ იყოს ნაცნობი და ამიტომ **404.html** გვერდს ამოგივგდებს, ამისგან თავის ასარიდებლად, **404.html** გვერდზეც იგივეს წერენ, რასაც **index.html**-ზე, ან ანგულარი გვთავაზობს უკეთესს მეთოდს:

**app-routing.module.ts**-ში **forRoot**-ს ვაწვდით მეორე არგუმენტს:

1. @NgModule({
2. imports:[
3. RouterModule.forRoot(appRoutes, {useHash: true})
4. ],
5. exports: [RouterModule]
6. })

აღნიშნული როუტს **#** სიმბოლოს ჩაუმატებს.  
**#-ს ეუბნება სერვერს, რომ მხოლოდ #-მდე არსებულ კონტენტს მიაქციოს ყურადღება, მის შემდეგ რაც წერია, ის დაიგნორდება**

**13. Understanding Observables**

169. Module Introduction

**Observables (introduction):**

იგი შეიძლება ჩაითვალოს, როგორც მონაცემების წყარო, იგი მოიცავს მაგალითად: **events, http requests, triggered in code,...**

არსებობს: **Observable**და Observer, მათ შორის კი თაიმლაინია, რომელზეც გვაქვს observable-ს მონაცემები.

**Observer**- იგი შეიძლება ითქვას ჩვენი კოდიც კი არის. 3 გზა არსებობს მონაცამებთან სამუშაოდ: **handle data (normal), handle error, handle completion** <- აქ ეგზეკუტირდება ცვენი კოდი.

შენ უნდა გადაწვიტო, რა მოხდეება, როცა მონაცემს მიიღებ, ან ერორს ან observable დაასრულებს მუშაობას. (ყველა **obsevable**არ სრულდება).

**13. Understanding Observables**

170. Analyzing Angular Observables

**Observables in Routing:**

**params**, რომელსაც **subscribe**-ს ვაბამთ, წარმოადგენს **observable**-ს

**13. Understanding Observables**

171. Getting Closer to the Core of Observables

**Observable general look:**

ერთ-ერთი **observable**, რომელიც შეგვიძლია **rxjs**-დან დავაიმპორტოთ არის: **interval**.

შემდეგ:

1. ngOnInit(){
2. interval(1000).subscribe(
3. count => { console.log(count }
4. )
5. }

იგი, როდესაც იმ კომპონენტზე გადავალთ, სადაც ესაა დეკლარირებული, კონსოლ ლოგში ამოგვიბეჭდავს ინქრემენტალ რიცხვებს, თუმცა შემდგომ სხვა გვერდზე თუ გადახვალ, იგი არ შეტყვეტს ამობეჭდვას, რადგან **observable**მაგას ყურადღებას არ აქცევს.

მის გასაჩერებლად:

აღნიშნული ობზერვებლის რათურნი უნდა შევინახოთ ცვლადში და შემდგომ **ngOnDestroy**-ით დავაანსუბსქრაიბოთ:

მაგალითად, ცვლადი, რომელშიც მას შევინახავთ არის: **unsubscribe**.

1. private unsubscribe: Subscription (უნდა დაიმპორტდეს)
2. ngOnDestroy(){
3. this.unsubscribe.unsubscribe()
4. }

**13. Understanding Observables**

172. Building a Custom Observable

**Observables create:**

წინაზე გამოვიყენეთ უკვე არსებული **observable**. ახლა შეგვიძლია ჩვენითაც შევქმნათ იგივე:

ამისთვის უნდა დავაიმპორტოთ **Observable rxjs**-დან და:

1. const costumIntervalObservable = Observable.create(observer=>{
2. let count =0;
3. setInterval(()=>{
4. observer.next(count);
5. count++
6. },1000)
7. })

   აქ: **observer**იღებს შემდეგ მნიშნველობებს: **next(), error(), complete()**

შემდეგ კი მაგას ჩვეულებრივ გამოვიძახებთ:

1. this.firstObsSubscribtion = costumIntervalObservable.subscribe(data => {
2. console.log(data)
3. })

**13. Understanding Observables**

173. Errors & Completion

**Observables errors:**

1. const costumIntervalObservable = Observable.create(observer=>{
2. let count =0;
3. setInterval(()=>{
4. observer.next(count);
5. if(count>3){
6. observer.error(new Error('couunt is greater 3!'))
7. }
8. count++
9. },1000)
10. })

როცა **observable**ერორს ამოაგდებს, აღარ გააგრძელებს მუშაობას.

ჰენდლინგი კი subscribe-დან ხდება:

1. this.firstObsSubscribtion = costumIntervalObservable.subscribe(data => {
2. console.log(data)
3. }, error => {
4. console.log(error)
5. alert(error.message)
6. })
7. }

**13. Understanding Observables**

173. Errors & Completion

**Observables completion:**

1. const costumIntervalObservable = Observable.create(observer=>{
2. let count =0;
3. setInterval(()=>{
4. observer.next(count);
5. if(count == 2) {
6. observer.complete()
7. }
8. if(count>3){
9. observer.error(new Error('couunt is greater 3!'))
10. }
11. count++
12. },1000)
13. })

კომპლიტის შემდეგ აღარაფერი აღარ მოხდება. მისი ჰენდლინგისთვის **subscribe**-ში ვამატებთ შემდეგნაირ არგუმენტს:

1. ()=> {
2. console.log('Completed!')
3. }

როცა დასრულებულია, ცალკე **unsubscribe**აღარაა საჭირო

**13. Understanding Observables**

175. Understanding Operators

**Observables Operators:**

**operators**დგებიან **observable**-ს და **subscription**-ს შორის და მაგალითად, ფილტრავენ კონკრეტულ მონაცემებს, რომელიც საბოლოოდ **Observer**-ს გადაეცემა.

მათ გამოსაყენებლად, მაგალითად, შეგვიძლია იმ ცვლადს, რომელშიც observable ინახება, მივაბათ მეთოდი**.pipe().**

შემდეგ **observable**უნდა დავაიმპორტოთ **rxjs/operators**-დან, სადაც მრავალი ოპერატორია.

მაგალითად, თუ დავაიმპორტებთ **map**-ს, იგი ჩაჯდება შემდეგ **.pipe()**-მეთდში:

1. this.firstObsSubscribtion = costumIntervalObservable.pipe(map((data:number)=>{
2. return 'Round ' + (data+1);
3. })).subscribe...

**13. Understanding Observables**

176. Subjects

**Observables Subject:**

მაგალითად სერვისში შეგვიძლია დავაიმპორტოთ **Subject rxjs**-დან.

იგი **EventEmitter**-ის მსგავსად იწერება საწყისში, ოღონდ **emit**-ის ნაცვლად იწერება **next**.

Subject ჩვეულებრივი **observable**-სგან იმით განსხვავდება, რომ**next()** მეთოდს შეგიძლია გარედანაც აქტიურად დაუძახო, განსხვავებით **observable**-სგან, სადაც მხოლოდ შიგნით გამოიძახებოდა.

1. activatedEmitter = new Subject<boolean>()
2. this.userService.activatedEmitter.next(true)

რეკომენდირებულია, რომ **EventEmitter**-ს ნაცვლად Subject გმაოიყენო. მათ ასევე შეგიძლია **unsubscribe**გაუკეთო:

სადაც დასუბსქრაიბებული გაქვს **Subject**, სტანდარტული სინტაქსით დააანსუბსქრაიბებ.

1. ngOnDestroy(){
2. this.activatedSub.unsubscribe()
3. }

\*შენიშნვა: Subject-ს ვერ გამოვიყენებთ @Output-თან. მხოლოდ ქროსკომპონენტებთან.

**15. Handling Forms in Angular Apps**

183. Template-Driven (TD) vs Reactive Approach

**Angular forms intro:**

ანგულარი გვთავაზობს ფორმებთან მიდგომის ორ გზას:

**1. Template-Driven** - ანგულარი ქმნის ფორმის ობიექტს **DOM**-დან

**2. Reactive**- ლოგიკის ნაწილში ქმნი ფორმს და შემდეგ სინქრონიზებას უკეთებ **DOM**-თან

**Forms TD:**

ანგულარის ფორმსების გამოსაყენებლად **FormsModule**უნდა დავაიმპორტოთ **@angular/forms**-დან

იმისათვის, რომ ანგულარმა აღმოაჩინოს ფორმსი, უნდა დაამატო controls თითოეულ ინფუთს: **ngModel და name="რაც გსურს"**

1. <input type="text" id="username" class="form-control" ngModel name="username

**15. Handling Forms in Angular Apps**

186. TD: Submitting and Using the Form

**Forms TD Submit:**

საბმიტისას **form**-ის თაგში უნდა ჩავამატოთ **(ngSubmit)** და ასევე ლოქალ რეფერენსი: **#f**, შემდეგ კი აღნიშნული ჩავაწოდოთ საბმიტის მეთოდის არგუმენტად. ლოგიკის ნაწილში კი, ამ კონკრეტული არგუმენტის ტიპი უნდა იყოს: **HTMLFormElement**:

1. <form (ngSubmit)="onSubmit(f)" #f>
2. ...
3. </form>

**ts:**

1. onSubmit(form: HTMLFormElement){
3. }

ლოქალ რეფერენსის გატოლებას შეგვიძლია სხვა ინფორმაციაზე, მაგალითად**: #f="ngForm"** - ეს ანგულარს მიმართავს, რომ ფორმსზე ჰქონდეს წვდომა, რომელის ანგულარმა შექმნა. ამ შემთხვევაში, **HTMLFormElement**-ის ნაცვლად **NgForm**-ის ტიპი იქნება აღნიშნული არგუმენტი:

1. <form (ngSubmit)="onSubmit(f)" #f="ngForm">
2. ...
3. </form>

**ts:**

1. onSubmit(form: NgForm){
3. }

**15. Handling Forms in Angular Apps**

188. TD: Accessing the Form with @ViewChild

**Forms TD (ViewChild):**

წინა ქმედების ალტერნატივაა, **ViewChild**-ის გამოყენება. **html**-ში იგივე რჩება ყველაფერი, **ts**:

1. @ViewChild('f') form: NgForm;
2. onSubmit() {
3. console.log(this.form)
4. }

**15. Handling Forms in Angular Apps**

189. TD: Adding Validation to check User Input

**Forms TD (Validation) 1.1:**

**email**- ასევე დირექტივია, რომელიც ამოწმებს იმეილია თუ არა შეყვანილი ინფუთში.

ვალიდაციისას, ანგულარი ავტუმატურად ამატებს კლასებს და ასევე შეჰყავს ეს ინფორმაცია დაბრუნებულ ობიექტშიც, რომელსაც სამოამვლოდ შეგვიძლია მივწვდეთ.

**ჩაშენებული ვალიდაციები:**

https://angular.io/api/forms/Validators

**ჩაშენებული დირექტიული ვალიდატორები:**

https://angular.io/api?type=directive

ასევე შეგვიძლია ჩავამატოთ **html**-ის ვალიდაცია, რომელსაც ანგულარი ავტომატურად თიშავს:

ngNativeValidate - დირექტივის დამატებით

**15. Handling Forms in Angular Apps**

191. TD: Using the Form State

**Forms TD (Validation) 1.2:**

**html**თემფლეითში შეგვიძლია თავისუფლად გვქონდეს წვდომა ლოქალ რეფერენსზე, ამ შემთხვევაში **#f**-ზე, რომელიც ინახავს ფროფერთის - **valide**, შესაბამისად, ფროფერთი ბაინდინგით შეგვიძლია მაგალითად ღილაკის დაეისბლება თუ ვალიდური არაა:

<button [disabled]="!f.valid"> Submit </button>

**css**კლასების შეცვლა:

**CSS**-ში **ng-invalid**-ს გავუწეროთ კლასი (იგი ავტომატურად ემატება ანგულარის მიერ, როგორც წინაზე ვახსენე), მაგალითად:

1. input.ng-invalid.ng-touched{
2. border: 1px solid red;
3. }

**15. Handling Forms in Angular Apps**

192. TD: Outputting Validation Error Messages

**Forms TD (Validation) 1.3:**

ასევე თუ გვსურს რაიმე კონტენტის გამოჩენა ვალიდურობის შემოწმებისას, სასურველია კონკრეტულ ინფუთს ან სხვა ტიპს, ლოქალ რეფერენსი გავუწეროთ, რომელიც გაუტოლდება **ngModel**-ს, მაგალითად: <input   #email="ngModel"  />. ხოლო, გამოსაჩენ კონტენტში: <span \*ngIf="email.valid && email.touched" >Please enter valid email</span>

**15. Handling Forms in Angular Apps**

193. TD: Set Default Values with ngModel Property Binding

**Forms TD (default values):**

ამ შემთხვევაში შეგვიძლია **ngModel**შემდეგნაირად გამოვიყენოთ: [ngModel]="აქ წავა დეფაულტი"

**15. Handling Forms in Angular Apps**

194. TD: Using ngModel with Two-Way-Binding

**Forms TD (ngModel with two-way-binding):**

იგი ჩვეულებრივ გამოიყენება **ngModel**-თან, ასევე იგი დასაბმიტებისას ერთ-ერთი მნიშნევლობაც იქნება, მაგალითად:

**html:**

1. <div class="form-group">
2. <textarea
3. name="questionAnswer"
4. rows="3"
5. [(ngModel)]="answer"
6. class='form-control'
7. ></textarea>
8. </div>
10. <p>Your reply: {{answer}}</p>

**ts:**

1. answer = '';

**15. Handling Forms in Angular Apps**

195. TD: Grouping Form Controls

**Forms TD (Grouping Controls):**

იმისათვის, რომ ფორმის მონაცემები დავაჯგუფოთ, შეგვიძლია კონკრეტული ჯგუფის **div**-ს მაგალითად, გავუწეროთ ატრიბუტული დირექტივი: **ngModelGroup**, რომელიც მას დააჯგუფებს და ასევე გავუტოლოთ სტრინგს, მაგალითად: **ngModelGroup="userData"**, რომელიც **userData**-ს სახელის ქვეშ მოაქცევს მთელს ჯგუფს.

ასევე, როგორც წინაზე გავაკეთეთ, კონკრეტული ელემენტების ჯგუფის შემკრებ **div**-ზე ასევე შეგვიძლია ჩავამატოთ ლოქალ რეფერენსი: **#userData="ngModelGroup",** რომელიც კონკრეტული ჯგუფის ობიექტზე მოგვცემს წვდომას.

**15. Handling Forms in Angular Apps**

196. TD: Handling Radio Buttons

**Forms TD (Radio buttons):**

ამ შემთხვევაში შეგვიძლია რადიოს ინფუთ ელემენტს value გადავუკეთოთ ფროფერთი ბაინდინგად **([value])** და გავუტოლოთ რაიმე მნიშნელობას, მაგალითად **html**:

1. <div class="radio" \*ngFor="let gender of genders">
2. <label>
3. <input
4. type="radio"
5. name="gender"
6. ngModel
7. [value]="gender"
8. >
9. {{gender}}
10. required
11. </label>
12. </div>

**ts:**

1. genders = ['male', 'female'];

**15. Handling Forms in Angular Apps**

197. TD: Setting and Patching Form Values

**Forms TD (ფუნქციით წვდომა ფორმსზე) 1.1:**

მისი ერთი გზა არსებობს, მაგალითად **ts:**

1. @ViewChild('f') form: NgForm;
3. suggestUserName() {
4. const suggestedName = 'Superuser';
5. this.form.setValue({
6. userData:{
7. username: suggestedName,
8. email:'',
9. },
10. secret: 'pet',
11. questionAnswer: '',
12. gender: 'male'
13. })
14. }

**html:**

1. <button class="btn btn-default" type="button" (click)="suggestUserName()">Suggest an Username</button>

სადაც, ზუსტად ფორმსის წყობის მიხედვით უნდა მივწვდეთ ფორმსის ობიექტს და ისე შევცვალოთ. მისი**დიდი ნაკლია:** ის ყოველ ჯერზე გადააწერს დაკლიკებისას ობიექტში ჩაწერილ ინფორმაციას.

**15. Handling Forms in Angular Apps**

197. TD: Setting and Patching Form Values

**Forms TD (ფუნქციით წვდომა ფორმსზე) 1.2:**

მისი მეორე გზაა იგივე **Patching**, რომელიც ხელმისაწვდომია თავად **form**ატრიბუტში, მაგალითად:

1. @ViewChild('f') formInfo: NgForm;
2. suggestUserName() {
3. const suggestedName = 'Superuser';
4. this.formInfo.form.patchValue({
5. userData:{
6. username:suggestedName
7. }
8. })

**patchValue()** - გამოიყენება ფორმის კონკრეტულ მონაკვეთზე გადასაწერად, წინასგან განსხვავებით.

*\*შენიშნვა: form-საც აქვს setValue()*

**15. Handling Forms in Angular Apps**

198. TD: Using Form Data

**Forms TD (ობიექტში გაერთიანება ინფორმაციის):**

**ლოგიკის ფაილში შემდეგნაირად გაიერთიანდება, ts:**

1. submitted = false;
2. user = {
3. username: '',
4. email: '',
5. secretQuestion: '',
6. answer: '',
7. gender: ''
8. }
9. onSubmit(){
10. this.submitted = true;
11. console.log(this.formInfo)
12. this.user.username = this.formInfo.value.userData.username
13. this.user.email = this.formInfo.value.userData.email
14. this.user.secretQuestion = this.formInfo.value.secret
15. this.user.answer = this.formInfo.value.questionAnswer
16. this.user.gender = this.formInfo.value.gender
17. }

**15. Handling Forms in Angular Apps**

199. TD: Resetting Forms

**Forms TD (Reseting Form):**

ამისთვის უბრალოდ **onSubmit()**-ს წინა დავალებიდან, დავამატებთ შემდეგს:

this.formInfo.reset()

***\*ასევე შესაძლებელია ამ მეთოდში ობიექტის ჩასმა, რომელიც მხოლოდ კონკრეტულ მნიშვნელობის დაარესეტებს***

**15. Handling Forms in Angular Apps**

201. Reactive: Setup

**Forms Reactive Intro:**

რეაქტიული ფორმები ლოგიკის ფაილში იქმნება, ამისთვის საჭიროა რაიმე ფროფერთიში მათი შენახვა და ტიპად უნდა ჰქონდეს: **FormGroup**, რომელსაც დავაიმპორტებთ **@angular/forms**-დან.

1. import { FormGroup } from '@angular/forms';
2. signupForm: FormGroup;

ასევე საჭიროა **appModule**-ში უნდა დავაიმპორტოთ **ReactiveFormsModule @angular/forms**-დან

***\*FormsModule****აღარაა საჭირო, რადგან ის გამოიყენება მხოლოდ****Template Driven****ფორმსისას.*

**15. Handling Forms in Angular Apps**

202. Reactive: Creating a Form in Code

**Forms Reactive Creating form 1.1:**

ფორმსის შექმნა რეკომენდირებულია ისეთ **life-cycle Hook**-ში, რომელიც თემფლეითამდე რენდერდება, ამისთვის გამოვიყენებთ **ngOnInit()**-ს.  new ქივორდის საშუალებით შევქმნით ახალ ფორმსს:

1. signupForm: FormGroup;
3. ngOnInit(){
4. this.signupForm = new FormGroup({})
5. }

ამ შემთხვევაში ფორმსი შევქმენით, თუმცა მას **ქონთროლსი**არ აქვს.

მათ შემოსატანად **FormsGroup**-ს ობიექტში უნდა ჩავაწოდოთ შემდეგნაირი ინფორმაცია:

1. this.signupForm = new FormGroup({
2. 'username': new FormControl(null),
3. 'email': new FormControl(null),
4. 'gender': new FormControl('male'),
6. })

**15. Handling Forms in Angular Apps**

203. Reactive: Syncing HTML and Form

**Forms Reactive Creating form 1.2:**

შემდეგ მივდივართ **html**-ში და პირველ რიგშ ფორმს ვაწვდით შემდეგ დირექტივს: **formGroup**, რომელიც დაუკავშირდება ჩვენს **ts**ფროფერთის, რომელშიც წინაზე შექმნილი ფორმსი ინახება:

1. <form [formGroup]="signupForm">
3. </form>

შემდეგ კი **ქონთროლებს**ვაკავშირებთ:

1. <form [formGroup]="signupForm">
2. <input type="text" formControlName="username" />
3. <input  type="text" formControlName="email" />
4. </form>

**ts:**

1. signupForm: FormGroup;
3. ngOnInit(){
4. this.signupForm = new FormGroup({
5. 'username': new FormControl(null),
6. 'email': new FormControl(null),
7. 'gender': new FormControl('male'),
8. })
9. }

**15. Handling Forms in Angular Apps**

204. Reactive: Submitting the Form

**Forms Reactive Creating form 1.3:**

**Submit**-ს დროს **TD**-ს მსგავსად, **form**-ს უნდა დავამატოთ **ngSubmit**და ჩავაწოდოთ საბმიტის მეთოდი, თუმცა ლოქალ რეფერენსი აღარაა საჭირო იმისათვის, რომ **ობიექტზე ინფორმაციას მივწვდეთ:**

1. <form [formGroup]="signupForm" (ngSubmit)="onSubmit()">
3. </form>

**ts:**

1. signupForm: FormGroup;
3. ngOnInit(){
4. this.signupForm = new FormGroup({
5. 'username': new FormControl(null),
6. 'email': new FormControl(null),
7. 'gender': new FormControl('male'),
8. })
9. }
10. onSubmit(){
11. console.log(this.signupForm)
12. }

**15. Handling Forms in Angular Apps**

205. Reactive: Adding Validation

**Forms Reactive Validating 1.1:**

არ უნდა დაგვავიწყდეს, რომ რეაქტივ ფორმსის შემთხვევაში თემფლეითიდან მხოლოდ სინქრონიზება ხდება ფორმის, კონფიგურირება მთლიანად **ts** ფაილზეა გადასული.

ამისთვის **ts**ფაილში, სადაც ფორმსი გვაქვს შექმნილი, თითოეულ **FormControl**-ს მეორე არგუმენტად შეგვიძლია ჩავაწოდოთ როგორც ერთი, ასევე ვალიდატორების მთელი მასივი, ამისთვის უბრალოდ **Validators**-ის დაიმპორტებაა საჭირო **@angular/forms**-დან:

1. this.signupForm = new FormGroup({
2. 'username': new FormControl(null, Validators.required),
3. 'email': new FormControl(null, [Validators.required, Validators.email]),
4. 'gender': new FormControl('male'),
5. })

**15. Handling Forms in Angular Apps**

206. Reactive: Getting Access to Controls

**Forms Reactive Validating 1.2:**

როგორც **TD**-ში შეგვეძლო ვალიდაციიისას მაგალითად რაიმე ტექსტის გამოჩენა, რეაქტის დროს აღნიშნული ცოტა სხვანაირად ხდება:

1. <input
2. type="text"
3. id="username"
4. formControlName="username"
5. class="form-control" />
6. <span \*ngIf="!signupForm.get('username').valid && signupForm.get('username').touched" class="help-black">Please enter a valid username!</span>

სადაც იმ ცვლადს ვწვდებით, რომელშიც ფორმია და **get()** მეთოდით ვიღებთ მასზე ინფორმაციას, რაზეც **.valid** თავისუფლად მაგრდება.

თუ ზოგადად ფორმის ვალიდაცია გვსურს და არა მხოლოდ რომელიმე კონტროლერის, მაშნ **get()**მეთოდი საჭირო აღარაა:

1. <span \*ngIf="!signupForm.valid && signupForm.touched" class="help-black">Please enter a valid Data!</span>

**15. Handling Forms in Angular Apps**

207. Reactive: Grouping Controls

**Forms Reactive კონტროლერების დაჯგუფება:**

**Grouping controls TD**-შიც გვქონდა, თუმცა აქ სხვანაირად ხდება:

აღნიშნული ქმედება, თავისთავად, ისევ **ts**ფაილში მიდის:

**FormGroup**-ის ობიექტში ახალი **FormGroup**ობიექტი ემატება *(ლოგიკურია)*:

1. this.signupForm = new FormGroup({
2. 'userData': new FormGroup({
3. 'username': new FormControl(null, Validators.required),
4. 'email': new FormControl(null, [Validators.required, Validators.email])
5. }),
6. 'gender': new FormControl('male'),
7. })

თუმცა ამ შემთხვევაში, ეს რომ თემფლეითშიც გამოვიყენოთ, კონკრეტული დაჯგუფების სისტემის მსგავსად, იქაც უნდა ჩავსვათ კონკრეტული კონტროლები გამოსაცალკევებელ **DIV**-ში, რომელსაც ფროფერთიდ გაეწერება: **formGroupName="აქ იქნება ჯგუფის სახელი, მაგ. userData",**ასევე შესაბამისად უნდა განვაახლოთ ვალიდაციიები get()-ში სწორი **path**-ის მითითებით **('userData.username')**.

**15. Handling Forms in Angular Apps**

209. Reactive: Arrays of Form Controls (FormArray)

**Forms Reactive (Form Arrays) 1.1:**

ჩვენ ასევე შეგვიძლია გვქონდეს კონტროლების მასივი, რომელშიც, მაგალითად, მომხმარებელი თავად დააფუშინგებს ახალ ფორმკონტროლებს, ამისათვის ჯერ **ts**ფაილში **new**ქივორდით უნდა დავამატოთ ახალი ფროფერთი ჩვენს **FormGroup**-ს, რომლის ტიპიც იქნება**FormArray([])**:

'hobbies': new FormArray([])

**FormArray**უდნა დავაიმპორტოთ **@angular/forms**-დან.

შემდეგ შეგვიძლია რომელიმე მეთოდს მივაბათ ამ კონკრეტულ მასივში ახალი კონტროლების ჩაყრის ფუნქცია შემდეგნაირად:

1. onAddHobby(){
2. const control = new FormControl(null, Validators.required);
3. (<FormArray>this.signupForm.get('hobbies')).push(control);
4. }

**15. Handling Forms in Angular Apps**

209. Reactive: Arrays of Form Controls (FormArray)

**Forms Reactive (Form Arrays) 1.2:**

**html**-ში კი:

1. <div formArrayName="hobbies">
2. <h4>Your Hobbies</h4>
3. <button class="btn btn-default" type="button" (click)="onAddHobby()">Add Hobby</button>
4. <div class="form-group" \*ngFor="let hobbyControl of signupForm.get('hobbies').controls; let i = index">
5. <input type="text" class="form-control" [formControlName]="i">
6. </div>
7. </div>

**15. Handling Forms in Angular Apps**

210. Reactive: Creating Custom Validators

**Forms Reactive (create validator):**

საკუთარი ვალიდატორის შესაქმნელად, ლოგიკაში ჯერ უნდა შევქმნათ ღნიშნული ვალიდაციის ფუნქცია, მაგალითად ვაკონტროლებთ, რა ტიპის ინფუთი არ შეიძლება, რომ იყოს. **ts**:

1. forbiddenUsernames = ['Chris', 'Anna'];
2. ngOnInit(){
3. this.signupForm = new FormGroup({
4. 'userData': new FormGroup({
5. 'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),
6. 'email': new FormControl(null, [Validators.required, Validators.email])
7. }),
8. 'gender': new FormControl('male')
9. })
10. }
12. forbiddenNames(control: FormControl):{[s:string]:boolean} {
13. if(this.forbiddenUsernames.indexOf(control.value) !== -1){
14. return {'NameIsForbidden': true};
15. }
16. return null;
17. }

**15. Handling Forms in Angular Apps**

211. Reactive: Using Error Codes

**Angular forms Error:**

ანგულარი ინდივიდუალურად ამატებს ობიექტებს **Error**-კოდებს.

აღნიშნული კოდები შეიძლება გამოყენებულ იქნას ერორ მესიჯის გამოტანისას **html**-ში. მაგალითად, წინაზე შექმნილი ვალიდეშნზე დაყრდნობით **html**:

1. <div class="form-group">
2. <label for="username">Username</label>
3. <input type="text" id="username" formControlName="username" class="form-control">
5. <span \*ngIf="!signupForm.get('userData.username').valid && signupForm.get('userData.username').touched" class="help-black">
7. <span \*ngIf="signupForm.get('userData.username').errors['NameIsForbidden']">This name is invalid</span>
9. <span \*ngIf="signupForm.get('userData.username').errors['required']">
10. This name is invalid</span> This field is required!
11. </span>
12. </div>

**15. Handling Forms in Angular Apps**

212. Reactive: Creating a Custom Async Validator

**Forms Reactive (create Async validator):**

ასევე შეგვიძლია **ასინქრონული ვალიდატორიც** შევქმნათ. წინაზე შექმნილ ვალიდეიშენზე დაყრდნობით. **ts:**

1. ngOnInit(){
2. this.signupForm = new FormGroup({
3. 'userData': new FormGroup({
4. 'username': new FormControl(null, [Validators.required, this.forbiddenNames.bind(this)]),
5. 'email': new FormControl(null, [Validators.required, Validators.email], this.forbiddenEmails)
6. }),
7. 'gender': new FormControl('male'),
8. 'hobbies': new FormArray([])
9. })
10. }
12. forbiddenEmails(control: FormControl):Promise<any> | Observable<any> {
13. const promise = new Promise<any>((resolve,reject)=> {
14. setTimeout(()=>{
15. if(control.value === 'test@test.com'){
16. resolve({'emailIsForbidden': true})
17. }else{
18. resolve(null)
19. }
20. } ,1500)
21. })
22. return promise
23. }

**15. Handling Forms in Angular Apps**

213. Reactive: Reacting to Status or Value Changes

**Forms Reactive (track form state):**

სტეიტზე თრაქინგი, ანუ **ასინქრონულის**მაგალითზე, რა **მდგომარეობაშია ფორმსი**, მის მიხედვით რომ ვიმოქმედოთ.

ამისთვის გამოიყენება მაგალითად 2 **Observable**:

**valueChanges**და **statusChanges**, რომლებიც ჩვენს ფორმსზე მაგრდებიან:

1. this.signupForm.valueChanges.subscribe(
2. (value)=> console.log(value)
3. )
5. this.signupForm.statusChanges.subscribe(
6. (value)=> console.log(value)
7. )

**15. Handling Forms in Angular Apps**

214. Reactive: Setting and Patching Values

**Forms Reactive (Setting and Patching values):**

როგორც **TD**ფორმსში, აქაც იგივენაირად ხდება მსგავსი წვდომა, ასევე **.reset()**მეთოდიც იგივენაირად გამოიყენება:

1. this.signupForm.setValue({
2. 'userData':{
3. 'username': 'Max',
4. 'email':'max@test.com'
5. },
6. 'gender': 'male',
7. 'hobbies': []
8. });
10. this.signupForm.patchValue({
11. 'userData':{
12. 'username': 'Max',
13. 'email':'max@test.com'
14. }
15. });

**16. Course Project - Forms**

237. Deleting Ingredients and Some Finishing Touches

**Deleting all Items in a FormArray**

As of **Angular 8+**, there's a new way of **clearing all items**in a FormArray.

1. (<FormArray>this.recipeForm.get('ingredients')).clear();

The clear() method automatically loops through all registered FormControls (or FormGroups) in the FormArray and removes them.

It's like manually creating a loop and calling removeAt() for every item.

**Pipes Introduction:**

იგი ანგულარში ჩაშენებული ხელსაწყოა, რომელიც საშუალებას გვაძლევს აუთფუთის ტრანსფორმაციის.

სხვადასხვა ტიპის პაიპებ არსებობს: მაგალითად **სინქრონული და ასინქრონული.**

**17. Using Pipes to Transform Output**

240. Using Pipes

**Pipes First using:**

პაიპების ერთადერთი გამოსაყენებელი სივრცე **თემფლეითია**;

პაიპების გამოსაყენებლად სტრინგ ინტერფოლაციაში ვამატებთ

**|** და შემდეგ პაიპის სახელს:

1. {{ server.instanceType | uppercase }}
2. {{ server.started | date }}

**17. Using Pipes to Transform Output**

241. Parametrizing Pipes

**Pipes Configuration:**

ამისთვის პაპს **:-**ს ვამატებთ:

1. {{ server.started | date:'fullDate' }}

**17. Using Pipes to Transform Output**

242. Where to learn more about Pipes

**Pipes Library:**

https://angular.io/api?query=pipe

**17. Using Pipes to Transform Output**

243. Chaining Multiple Pipes

**Pipes ერთმანეთთან დაკავშირება:**

ამისთვის უბრალოდ ახალ **|** - ამ ნიშანს ვამატებთ და ახალ პაიპის ფროფერთის. მნიშნველოვანია ვიცოდეთ, რომ პაიპები **მარცხნიდან მაჯვნივ** იკითხება, ამიტომაც რიგითობას დიდი მნიშვნელობა აქვს *(არაწროად დაწერილმა შეიძლება დაბაგოს).*

**17. Using Pipes to Transform Output**

244. Creating a Custom Pipe

**Pipes Create (შექმნა):**

ამისთვის ვქმნით ცალკე ფაილს, ვადეკლარირებთ კლასს, რომელიც **implemets PipeTransform** (რომელიც უნდა დაიმპორტდეს **@angular/core**-დან). ასევე მნიშნველოვანია **@Pipe({})** დეკორატორის დამატებაც, სადაც შიგნით name დადეკლარირდება, რომელიც მიიღებს პაიპის სახელს.

აღნიშნული იმპლემენტი ითხოვს **transform()** მეთოდის დადეკლარირებას კლასში, რომელიც იღებს არგუმენტებს. ამის შემდეგ კი აღნიშნული კლასი უნდა დავაიმპორტოთ **appModule**-ის **declarations**-ში:

1. import { PipeTransform } from "@angular/core";
3. @Pipe({
4. name: 'shorten'
5. })
7. export class ShortenPipe implements PipeTransform {
8. transform(value:any){
9. if(value.length){
10. return value.substr(0,10) + '...';
11. }
12. return value
13. }
14. }

**html:**

1. <strong>{{ server.name | shorten }}</strong>

**17. Using Pipes to Transform Output**

245. Parametrizing a Custom Pipe

**Pipes Create & Custom (შექმნა და ფორმირება):**

ფორმირებისთვის **transform()**მეთოდს მეორე არგუმენტად ჩავაწვდით უკვე ამ კონკრეტულ პარამეტრს, რაც შეიძლება ორწერტილის მერე თემფლეითში ჩაიწეროს და შესაბამისად შეიცვალოს *(ყოველ ახალ პარამეტრზე თემფლეითში უბრალოდ****ორწერტილს****ჩავამატებთ):*

1. transform(value:any, limit: number){
2. if(value.length > limit){
3. return value.substr(0,limit) + '...';
4. }
5. return value
6. }

**html:**

**17. Using Pipes to Transform Output**

246. Example: Creating a Filter Pipe

**Pipes (filter pipe) 1.1:**

**CLI**-დან **Pipe**-ის შექმნა:ng g p [სახელი]

ამ შემთხვევაში **pipe**-ს გამოვიყენებთ ფილტრაციისთვის.

ჯერ ვქმნით ახალ პაიპს **CLI**-დან. შემდეგ კი**ts:**

1. transform(value: any, filterString: string, propName:string): any {
2. if(value.length === 0 || filterString === ''){
3. return value;
4. }
5. const resultArray = [];
6. for(const item of value){
7. if(item[propName] === filterString){
8. resultArray.push(item);
9. }
10. }
11. return resultArray
12. }

**17. Using Pipes to Transform Output**

246. Example: Creating a Filter Pipe

**Pipes (filter pipe) 1.2:**

აქ აღსანიშნავია, რომ პაიპი შეიძლება დაიმტოს თავად **\*ngFor**სტრუქტურულ დირექტივსაც:

1. \*ngFor="let server of servers | filter: filteredStatus: 'status'"

აღნიშნულ ინფორმაციას კი იღებს აქედან:

1. <input type="text" [(ngModel)]="filteredStatus">

**filteredStatus**დადეკლარირებულია კომპონენტის **ts**ფაილში

**17. Using Pipes to Transform Output**

247. Pure and Impure Pipes (or: How to "fix" the Filter Pipe)

**Pipes filter issue:**

როდესაც გაფილტვრის პაიპს ავაწყობთ ერთი პრობლემა იქმნება, იგი ყოველ ჯერზე, როდესაც ინფორმაცია განახლდება კონკრეტულ ობიექტში, არ ანახლებს პაიპს, რადგან აღნიშნული შესაძლოა დიდ პერფორმანსად დაგვიჯდეს, თუმცა ფორსირება შესაძლებელია **@Pipe**-ის დეკორატორში ახალი ფროფერთის **(pure: false)** ჩამატებით, რომელიც დეფაულტზე **true**არის:

1. @Pipe({
2. name: 'filter',
3. pure: false
4. })

**17. Using Pipes to Transform Output**

248. Understanding the "async" Pipe

**Pipes Async:**

მაგალითად თუ ვიყენებთ **ასინქრონულად**მონეცემების მიღებას (სერვერიდან), შეგვიძლია ამაში ასინქრონული პაიპი დავიხმაროთ.

მაგალითად, თუ ლოგიკის ფაილში **Promise**გვაქვს, რომელიც გარკვეული დროის შემდეგ აბრუნებს ინფორმაციას, შესაძლოა ანგულარმა თემფლეითი არ განაახლოს ამ გარკვეული დროის შემდეგ. ამის მოსაგვარებლად, სტრინგ ინტერპოლაციაში ვუმატებთ პაიპს: **async**, რომელიც ჩაშენებული პაიპია *(იგი ასევე კარგად მუშაობს****Observable****-თან)*:

1. <h2>App Status: {{appStatus | async}}</h2>

**18. Making Http Requests**

251. How Does Angular Interact With Backends?

**Angular DataBase:**

ანგულარი პირდაპირ არ უკავშირდება მონაცემთა ბაზას (სერვერს).

შენ გზავნი **Http Request**და იღებ **Http Response** (სერვერზე).

**18. Making Http Requests**

254. Sending a POST Request

**Http package 1.1:**

პირველ რიგში უნდა დავრეგისტრირდე **FareBase**-ზე, შევქმნა ახალი პროექტი და**Realtime Database**გამოვიყენო, სადაც ტესტმოდზე გავუშვებ.

**appModule**-ში უნდა შემოვიტანოთ **HttpClientModule**, რომელსაც დავაიმპორტებთ import { HttpClientModule } from '@angular/common/http'-დან.

შემდეგ **http client** უნდა დაინჯექტდეს **ts**ფაილის კონსტრუქტორში:

constructor(private http: HttpClient) {}

**HttpClient**დაიმპორტდებაimport { HttpClient } from '@angular/common/http'-დან.

შემდეგ **Firabase**-ში შექმნილ მონაცემთა ბაზის ლინკს ვიყენებთ შემდეგნაირად:

1. constructor(private http: HttpClient) {}
2. onCreatePost(postData: { title: string; content: string }) {
3. this.http.post('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json',postData)
4. }

**18. Making Http Requests**

254. Sending a POST Request

**Http package 1.2:**

წინა შემთხვევაში აღსანიშნავია, რომ**.json** დამატება მხოლოდ **FireBase**-ის მიერ მოითხოვება და არა თავად სერვერის.

ასევე, მეორე არგუმენტად რამდენადაც იმ ინფორმაციას აწვდი, რომელიც სერვერზე უნდა წავიდეს, იგი ავტომატურად ანგულარის მიერ პარსინგდება **json**ფორმატში.

ასევე მნიშნევლოვანია, რომ ვიცოდეთ: არანაირი რექვესტი არ გაიგზავნება ანგულარიდან, თუ **Observable**არაა ჩართული საქმეში, რადგან **Observable**აკვირდება პასუხს სერვერიდან. ამისთვის ჩვენს რექვესტს უნდა დავუმატოთ **subscribe()** მეთოდი:

1. onCreatePost(postData: { title: string; content: string }) {
2. this.http.post('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json', postData).subscribe(responseData => {
3. console.log(responseData)
4. })
5. };

**18. Making Http Requests**

255. GETting Data

**Http request GET:**

იმისათვის, რომ სერვერზე შენახული ინფორმაცია გამოვითხოვოთ:

ვქმნით მეთოდს, რომელსაც გამოვიძახევთ ხოლმე სამომავლოდ დანიშნულებისამებრ:

1. private fetchPosts(){
2. this.http.get('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json').subscribe(post => {
3. console.log(post)
4. })
5. }

**18. Making Http Requests**

256. Using RxJS Operators to Transform Response Data

**Http response GET მიღებული ინფორმაციის დამუშავება:**

ამისთვის გამოიყენება **Observable .pipe()** მეთოდის დამატებით, რომელიც არგუმენტად მიიღებს **.map()** მეთოდს, რომელიც იმპორტდება  **import {map} from 'rxjs/operators'**-დან:

1. private fetchPosts(){
2. this.http.get('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json').pipe(map(responseData =>{
3. const postsArray = [];
4. for(const key in responseData){
5. if(responseData.hasOwnProperty(key)){
6. postsArray.push({...responseData[key], id: key})
7. }
8. }
9. return postsArray
10. }))
11. .subscribe(post => {
12. // ... აქ შეგვეძლო ასევე ინფორმაციის დამუშავება, თუმცა უკეთესი პრაქტიკისთვის ზემოთ მოყვანილი ჯობს
13. console.log(post)
14. })
15. }

**18. Making Http Requests**

257. Using Types with the HttpClient

**Http Response GET ტიპების გამოყენება:**

მაგალითად შეგვიძლია შევქმნათ მოდელი, რომელიც იქნება ტიპი, რომელსაც აბრუნებს კონკრეტული თუნდაც მეთოდი ან არის არგუმენტი ეგეთი ტიპის. აღნიშნული გაკეთება შეგვიძლია შიგნითვე **map()** მეთოდში:

1. map((responseData: {[key: string]:Post}) => {}

ან თავად **.get()** მეთოდში (უკეთესია):

1. get<{ [key: string]: Post }>()

აღნიშნული ასევე რეკომენდირებულია გამოვიყენოთ **Post**მეთოდთან.

1. post<{ name: string }>()

**18. Making Http Requests**

258. Outputting Posts

**Http Response თემფლეითში გამოტანა ინფორმაციის:**

ამისთვის **get**-ის **subscribe**-ში უბრალოდ უნდა გადავიტანოთ ტრანსფორმირებული მიღებული ინფორმაცია უკვე არსებულ ფროფერთიში და თემფლეიტში გავლუპოთ:

1. loadedPosts: Post[] = [];
2. .subscribe(post => {
3. this.loadedPosts = post;
4. })

**html:**

1. <ul class="list-group" \*ngIf="loadedPosts.length >= 1">
2. <li class="list-group-item" \*ngFor="let post of loadedPosts">
3. <h3>{{post.title}}</h3>
4. <p>{{post.content}}</p>
5. </li>
6. </ul>

**18. Making Http Requests**

259. Showing a Loading Indicator

**Http Response ჩატვირთვის შეტყობინება:**

შეგვიძლაი ახალი ფროფერთ ჩავამატოთ, რომელიც ბულიან იქნება და თემფლეიტში **\*ngIf**-ის წყალობით გამოაჩენს რაიმეს.

მისი მნიშნველობა შეიცვლება **fetching**-ისას და შემდეგ უკვე **subscribe**-სას იგივე მდგომარეობას დაუბრუნდება:

1. private fetchPosts(){
2. this.isFetching = true
3. this.http.get().pipe().subscribe(post => {
4. this.isFetching = false
5. })
6. }

**18. Making Http Requests**

261. Services & Components Working Together

**Http Request to Service:**

სერვისშიც შეგვიძლია გადავიტანოთ ჩვენი რექვესტი, თავად სერვისში აღნიშნული მეთდი უბრალოდ **return**-ს გააკეთებს **get**მეთოდის ხოლო კონკრეტული კომპონენტი კი ამ სერვისიდან **.subscribe()** მეთოდით მიწვდება (ანუ აამუშავებს რექვესტს). ან ამის ნაცვლად,**.Subject()**-იც შეგვიძლია გამოვიყენოთ.

**18. Making Http Requests**

262. Sending a DELETE Request

**Http Request DELETE:**

იგი მარტივად ხდება:

1. deletePosts(){
2. return this.http.delete('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json').subscribe(()=> {})
3. }

**18. Making Http Requests**

263. Handling Errors

**Http Request ERRORS 1.1:**

იმისათვის, რომ აღნიშნული გავტესტოთ, **firebase**-ის **realtime database**-ში **rules**-ში უნდა შევცვალოთ, მაგალითად**'read': false.**

ყველა **Observable**-ს, რამდენადაც ვიცით, აქვს **error**ჰანდლინგი, რომელსაც ამ შემთხვევაში გამოვიყენებთ და შესაბამისად გადავაკეთებთ, როგორც გვსურს:

1. ngOnInit() {
2. this.isFetching = true;
3. this.postsService.fetchPosts().subscribe( posts => {
4. this.isFetching = false;
5. this.loadedPosts = posts;
6. }, error => {
7. this.error = error.error.error
8. console.log(error)
9. })
10. }

**18. Making Http Requests**

264. Using Subjects for Error Handling

**Http Request ERRORS 1.2:**

ერორის ჰენდლინგის **მეორე გზაც** არსებობს:

ჩვენს შემთხვევაში სერვისში შევქმნათ **new Subject<string>()**, რომელსაც **.next()** მეთოდს მისცემს თავად ობსერვებლის **error**ფუნქცია, შემდეგ კი ჩვენი კომპონენტიდან, უბრალოდ აღნიშნული **Subject**უნდა დავასუბსქრაიბოთ, თუმცა აღნიშნული სუბსქრაიბი უნდა შევინახოთ ფროფერთიში, რომელსაც შემდეგ დავაანსუბსქრაიბებთ:

**service:**

1. error = new Subject<string>()
2. error => {this.error.next(error.message)}

**component.ts:**

1. private errorSub: Subscription;
2. ngOnInit() {
3. this.errorSub = this.postsService.error.subscribe(
4. errorMessage => {
5. this.error = errorMessage
6. }
7. )
8. }
9. ngOnDestroy(){
10. this.errorSub.unsubscribe();
11. }

**18. Making Http Requests**

265. Using the catchError Operator

**Http Request catchError:**

აღნიშნული კეთდება სერვისის მხარეს ჩვენს შემთხვევაში.

ამისთვის **catchError**უნდა დაიმპორტდეს **rxjs/operato**rs-დან და **throwError**უნდა დაიმპორტდეს **rxjs**-დან.

შემდეგ კი **.pipe(**) მეთოდშივე შეგვიძლია ასევე დავამატოთ **catchError()** მეთოდი რომელიც დააბრუნებს **throwError()**მეთოდს:

1. return this.http.get().pipe(map(), catchError(
2. (error => {
3. //send to analytics server
4. return throwError(error)
5. }
6. )
7. )

**18. Making Http Requests**

267. Setting Headers

**Http Request Headers:**

ყველა ტიპის რექვესტს აქვს მეორე ან მესამე არგუმენტად ე.წ. **Headers**, რომელიც აკონფიგურირებს რექვესტს:

აღნიშნული წარმოადგენს ობიექტს, პირველი შეიძლება იყოს თავად ფროფერთი **headers**, რომელიც იღებს მნიშნველობად new HttpHeaders({"Costum-Header": "Hello"}) (ობიექტის შიგნით უბრალო მაგალითია მოცემული).

**HttpHeaders**უნდა დაიმპორტდეს **@angular/common/http**-დან (მანდვე ზის ასევე **HttpClient**)

**18. Making Http Requests**

268. Adding Query Params

**Http Request Query Params:**

აღნიშნული, როგორც წინა, დამოკიდებულია მოცემული სერვერი უჭერს თუ არა მხარს კონკრეტულ პარამეტრებს.

ჩვენს შემთხვევაში, **firebase**მხარს უჭერს მაგალითად **'print'='pretty'**-ის.

მაგალითი:

იგი ასევე ემატება ჰედერების ობიექტში, ან პირდაპირ, ან ცალკე დეკლარირდება ცვლადი (ბევრი პარამეტრის შემთხვევაში) და ისე ხდება ესაინმენტი:

1. let searchParams = new HttpParams();
2. searchParams = searchParams.append('print','pretty');
3. searchParams = searchParams.append('costum', 'key'); //firebase ამას მხარს არ უჭერს
5. return  this.http.get<{ [ket: string]: Post }>('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json',
7. {
8. headers:new HttpHeaders({"Costum-Header": "Hello"}),
9. params:searchParams
10. }
12. )

**18. Making Http Requests**

269. Observing Different Types of Responses

**Http Different type of Responses 1.1:**

როდესაც ჩვეულებრივ რექვესტს ვაკეთებთ სერვერზე, მაგალითად  **POST request**-ს, ანგულარის სტანდარტული გამოძახებით, იგი მხოლოდ body-ს გვაწვდის. თუ გვსურს რექვესტზე მთელი ინფორმაცია მივიღოთ, მაშინ, მაგალითად ჩვენს **POST**რექვესტში ჩავამატოთ მესამე არგუმენტი, რომელიც წარმაოდგენს ჰედერს, ოღონდ ჰედერების ნაცვლად, მანდ შეგვიძლია ჩავწეროთ observe, რომლის დეფაულტ მნიშნელობა**'body'** არის, რომელიც გვაწვდის ბადის ინფორმაციას და ასევე ავტომატურად აპარსინგებს ჩაწოდებულ დათას. ამის ნაცვლად შეგვიძლია დავწეროთ **'response'**, რომელიც მთელ ინფორმაციას მოგვაწვდის რექვესტზე.

აღნიშნული დააბრუნებს მთლიან რექვესტზე ინფორმაციას, სადაც შემდეგ უკვე სუბსქრაიბიდან შეგვიძლია დეტალურად მივწვდეთ **body**-ს და ა.შ.

1. .subscribe(responseData => {
2. console.log(responseData.body)
3. }

**18. Making Http Requests**

269. Observing Different Types of Responses

**Http Different type of Responses 1.2:**

ასევე მას აქვს სხვა მნიშნველობაც, **'events'**, რომელიც კონკრეტულ სტატუსს ანიჭებს ნომერს, რომელიც შემდგომ შეგვიძლია შევადაროთ **HttpEventType**-ის რომელიმე ფროფერთის (ეს უნდა დავაიმპორტოთ **@angular/common/http**-დან). მის ისე გამოსაყენებლად, რომ ხელი არ შევუშალოთ **subscribe**-ს ან რომელიმე ფუნქციას, რომელიც **subscribe**-შია ჩაწოდებული, უნდა ვისარგებლოთ **tap()** ოპერატორით (**rxjs/operators**-დან):

1. deletePosts(){
2. return this.http.delete('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json',
3. {
4. observe: 'events'
5. })
6. .pipe(tap(event => {
7. console.log(event)
8. if(event.type === HttpEventType.Sent){
9. //..
10. }
11. if(event.type === HttpEventType.Response){
12. console.log(event.body)
13. }
14. }));
15. }

**18. Making Http Requests**

270. Changing the Response Body Type

**Http ბადის შეცვლა:**

დეფაულტზე ბადის უყენია **responseType: 'json'**, თუმცა აღნიშნული შეიძლება შევცვალოთ ჰედერებში:

1. return this.http.delete('https://httprequeststudy-default-rtdb.firebaseio.com/posts.json',
2. {
3. observe: 'events',
4. responseType: 'text'
5. }
6. )

თუმცა ობიექტთან მუშაობისას ჯობს **json**ტიპი გამოვიყენოთ.

**18. Making Http Requests**

271. Introducing Interceptors

**Http Interceptors 1.1:**

ისინი აკვირდებიან რექვესტებს და ყველა რექვესტს (თუ არ შეუზღუდავ) ამატებს დამატებით პარამეტრს ან თუნდაც ჰედერს.

მისთვის სასურველია ცალკე სერვისი შევქმნათ, რომელსაც **@Injectable()** არ ჭირდება. აღნიშნულის კლასი დააიმპლმენტებს **HttpInterceptor**-ს, რომელიც ასევე შემოდის **@angular/common/http**-დან (როგორც ყველაფერი ამ ფაილში ნახსენები.

შიგნით კი იქმნება მეთოდი**intercept()**, რომელიც იღებს ორ არგუმენტს: ერთის ტიპია: **HttpRequest<any>** და მეორის - **HttpHandler**.

**HttpRequest - იჭერს რექვესტს;**

**HttpHandler - უშვებს რექვესტს (უკვე შეცვლილს თუ დავურთავთ);**

აღნიშნულმა მეთოდმა უნდა დააბრუნოს **HttpHandler**, რომელსაც აქვს მეთოდი handle(), რომელსაც არგუმენტად ჩავაწვდით **HttpRequest**-ს.

appModule-ში კი provedrs-ში შემდეგნაირად უნდა დავაკავშიროთ ეს ინტერსეპტორი:

*(კოდი შემდეგში)*

**18. Making Http Requests**

271. Introducing Interceptors

**Http Interceptors 1.2:**

**ts service:**

1. import {  HttpHandler, HttpInterceptor, HttpRequest } from "@angular/common/http";
2. export class AuthInterceptorService implements HttpInterceptor {
3. intercept(req: HttpRequest<any>, next: HttpHandler ){
4. if(req.url){
5. }
6. console.log('Request is on its way')
7. return next.handle(req)
8. }
9. }

**appModule:**

1. providers: [
2. {
3. provide:HTTP\_INTERCEPTORS,
4. useClass: AuthInterceptorService,
5. multi: true
6. }
7. ]

**18. Making Http Requests**

272. Manipulating Request Objects

**Http Interceptors მანიპულაცია:**

ინტერსეპტორებში ასევე შესაძლებელია რიქვესტის მოდიფიცირება.

ამისთვის ახალი (კლონირებული) რექვესტი უნდა შევინახოთ ცალკე ცვლადში და იგი ჩავაწოდოთ შემდეგ **next.handle()**-ს არგუმენტად. მაგალითად:

1. import {  HttpHandler, HttpInterceptor, HttpRequest } from "@angular/common/http";
2. export class AuthInterceptorService implements HttpInterceptor {
3. intercept(req: HttpRequest<any>, next: HttpHandler ){
4. const modifiedRequest = req.clone({
5. headers: req.headers.append('Auth', 'xyz')
6. })
7. console.log('Request is on its way')
8. return next.handle(modifiedRequest)
9. }
10. }

**.clone({})**-ს შეიძლება ყველაფერი ჩავაწოდოთ. მაგალითად, url-ის შეცვლაც შეგვიძლია.

**18. Making Http Requests**

273. Response Interceptors

**Http Intercaptors მანიპულაცია პასუხზე:**

აღნიშნულში ასევე შეგვიძლია არა მხოლოდ გაგზავნილი რექვესტით მანიპულირება, არამედ მიღებულითაც. ამისთვის **handle()** მეთოდს უნდა მივაბათ **.pipe()** და შემდგომ რომელიმე ოპერატორით მოვახდინოთ მანიპულირება:

1. return next.handle(modifiedRequest).pipe(tap(event => {
2. console.log(event)
3. if(event.type === HttpEventType.Response){
4. console.log('Response arrived, bodyData: ')
5. console.log(event.body)
6. }
7. }))

**18. Making Http Requests**

274. Multiple Interceptors

**Http Intercaptors რამდენიმეს გამოყენება:**

თუ რამდენიმე ინტერსეტერს ვიყენებთ, მნიშნელოვანია providers-ში მათი სწორი თანმიმდევრობით დალაგება, რადგან აღნიშნული განსაზღვრავს რომელს წაიკითხავს პირველს და ა.შ.:

1. providers: [
2. {
3. provide:HTTP\_INTERCEPTORS,
4. useClass: AuthInterceptorService,
5. multi: true
6. },
7. {
8. provide:HTTP\_INTERCEPTORS,
9. useClass: LoggingIntercaptor,
10. multi: true
11. }
12. ]

**დოკუმენტაცია:**

*https://angular.io/guide/http*

**20. Authentication & Route Protection in Angular**

286. How Authentication Works

**Authentication introduction (how it works) 1.1:**

არსებობს ორი მხარე: **Client** და **Server**.

**Client**არის ჩვენი ბრაუზერი, ხოლო **Server**- სადაც მოთხოვნას ვაგზავნით.

აუთენტიფიკაციის გაწერა ჯავასკრიპტში არაა სასურველი, რადგან მასზე წვდომა თავისუფლად შეუძლია მომხმარებელმა რომ მიიღოს. ამისათვის უმჯობესია იგი **სერვერის მხარეს გაიწეროს**.

იმ შემთხვევაშ, თუ ძველ ტექნოლოგიას ვიყენებთ, სერვერსა და ჩვენს ბრაუზერს შორის მყარდება Sessions, რაც გულისხმობს ყოველ ჯერზე ახალი **html**ფაილის დარენდერებას.

ამჟამად, ანგულარი იყენებს არა Sessions, არამედ **RESTful API**-ს, რომლის დროსაც აუთენთიფიკაციისას სერვერზე როცა გაივლის მონაცემი ვალიდაციას, უკან იგზავნება **token**, ანუ **json**ფაილი, რომელიც შეიცავს მეტადეითას და იგი ენკოდირებულია, რაც იმას ნიშნავს, რომ ჩვენს აპლიკაციას შეუძლია გახსნას და შეინახოს მაგალითად ბრაუზერის სტორეიჯში და ყოველ ჯერზე, როცა რექვესტს გააკეთებს ამ სერვერზე, აღნიშნულ token-ს მიაბამს მაგალითად რექვესტის ჰედერში, რათა სერვერმა შეასრულოს მოთხოვნა.

**20. Authentication & Route Protection in Angular**

286. How Authentication Works

**Authentication introduction (how it works) 1.2:**

აღნიშნულ **token**-ს გააჩნია საიდუმლო გასაღები, რომელსაც მხოლოდ კონკრეტული სერვერი ხსნის, რადგან მის მიერაა შექმნილი. ასე არის დაცული **token**-ი.

ჩვენით ვერ შევძლებთ **token**-ის გენერირებას, რადგან იგი შესაბამისობაში არ მოვა სერვერზე არსებულ ალგორითმთან და უსაფრთხოების გასაღებთან და რექვესტი დაიბლოკება.

**20. Authentication & Route Protection in Angular**

290. Preparing the Backend

**Authentication FireBase SetUp:**

მის ასაწყობად პირველ რიგში სასურველია ცალკე კომპონენტი შევქმნათ, რომელსაც ასევე როუტინგშიც ჩავამატებთ.

მის გასამართად არაა საჭირო **firebase**-ის გამოყენება, რადგან შეგვიძლია ჩვენი ბაქი დავუწეროთ, თუმცა ამ ჯერზე **firebase**-ით ვისარგებლებთ, რადგან მანდ უკვე ყველაფერი მზადაა:

პირველრიგში, **firebase**-ის **database**-ში უნდა წავიდეთ და **rules**შევცვალოთ:

1. {
2. "rules": {
3. ".read": "auth != null",
4. ".write": "auth != null",
5. }
6. }

შემდეგ შევდივართ **authentication**-ში და ვააქტიურებთ*(მხოლოდ პირველს)* **Email/Password**-ს.

**20. Authentication & Route Protection in Angular**

292. Preparing the Signup Request

**Authentication SignUp 1.1:**

**დოკუმენტაცია**:*https://firebase.google.com/docs/reference/rest/auth*

შემდეგ სასურველია შევქმნათ აუთენტიკაციისთვის ცალკე სერვისი, რომელსაც დასჭირდება **HttpClient**-ის დაინჯექტება კონსტრუქტორში. შემდეგ მაგალითად ვქმნით **signUp()**მეთოდს და მანდედან ვაკეთებთ **POST**რექვესტს, რომელსაც **url**-ში ვაწვდით ლინკს, რომელიც დოკუმენტაციის ***Sign Up with email/password***-ის სექციაში ნახსენებ **Endpoint**-შია მითითებული: *https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=[API\_KEY]*

**API KEY** კი არის ჩვენს **firebase**-ექაუნთში **project view**-ს გვერდით არსებულ აიქონის **project setting**-ში, **Web Api Key**-ში.

როგორც დოკუმენტაციაშია, Post რექვესტის მეორე არგუმენტში სასურველია ჩავაწოდოთ ობიექტი, რომელიც მიიღებს **email, password** და **returnSecureToken**(რომელიც ყოველთვის true-ა) ფროფერთიებს და გაუტოლდებიან შესაბამის ვალიუს.

**20. Authentication & Route Protection in Angular**

292. Preparing the Signup Request

**Authentication SignUp 1.2:**

როგორც სტანდარტული **observable**, ამის ასამუშავებლადაც **subscribe**მეთოდი გვჭირდება. ამიტომ აღნიშნულს სასურველია სერვისში უბრალოდ **return**გავუწეროთ.

რამდენადაც აღნიშნული **POST**რექვესტი პასუხსაც მიიღებს **firebase**-ის სერვერიდან, აღნიშნულ პასუხს გარკვეული ფორმატი ექნება. მის დასადეკლარირებლად, შეგვიძლია **interface**შევქმნათ და **post**მეთოდს ჯენერიკ ტიპში ჩავაწოდოთ. დოკუმენტაციის მიხედვით, მსგავსი ტიპის რესპონსი მოვა (***intarface***-ში):

1. interface AuthResponseData {
2. kind: string,
3. idToken: string,
4. email: string,
5. refreshToken: string,
6. expiresIn: string,
7. localId: string
8. }

**20. Authentication & Route Protection in Angular**

292. Preparing the Signup Request

**Authentication SignUp 1.3:**

**signUp() მეთოდის სახე:**

1. signUp(email: string, password: string){
2. return this.http.post<AuthResponseData>('https://identitytoolkit.googleapis.com/v1/accounts:signUp?key=AIzaSyCLehbZXZikSTn\_nxZPOOJxwu60CqThr\_A',
3. {
4. email: email,
5. password: password,
6. returnSecureToken: true
7. })
8. }

**20. Authentication & Route Protection in Angular**

293. Sending the Signup Request

**Authentication SignUp 1.4:**

შემდეგ კი უბრალოდ აღნიშნულ სერვისს ჩვენს კომპონენტში დავაინჯექტებთ და შესაბამის მეთოდში დავასუბსქრაიბებთ:

1. onSubmit(form: NgForm){
2. const email = form.value.email;
3. const password = form.value.password;
4. this.authService.signUp(email, password).subscribe(res => {
5. console.log(res)
6. }, err => {
7. console.log(err)
8. })
9. }

**20. Authentication & Route Protection in Angular**

294. Adding a Loading Spinner & Error Handling Logic

**Authentication SignUp 1.5 (spinner & error):**

სპინერების საიტი: *https://loading.io/css/*

ერორს სტანდარტულად გავჰენდლავთ...

**20. Authentication & Route Protection in Angular**

295. Improving Error Handling

**Authentication SignUp 1.6 (spinner & error):**

ერორის ჰენდლინგი ასევე შეგვიძლია თავად სერვისშივე **catchError()** *(rxjs/operators-დან)*და **throwError()***(rxjs-სადნ)*-ის გამოყენებით:

**service:**

1. .pipe(catchError(errorRes =>{
2. let errorMessage = 'An unknown error occured!'
3. if(!errorRes.error || !errorRes.error.error){
4. return throwError(errorMessage);
5. }
6. switch(errorRes.error.error.message){
7. case 'EMAIL\_EXISTS':
8. errorMessage = 'This email exists already'
9. }
10. return throwError(errorMessage);
11. }))

**20. Authentication & Route Protection in Angular**

295. Improving Error Handling

**Authentication SignUp 1.7 (spinner & error):**

**component:**

1. ,errorMessage => {
2. console.log(errorMessage)
3. this.error = errorMessage
4. this.isLoading = false
5. }

**20. Authentication & Route Protection in Angular**

296. Sending Login Requests

**Authentication Sign in 1.1:**

შესვლისთვის იგივენაირად ვიქცევით თითქმის, უბრალოდ იგივე სერვისში ვქმნით სხვა მეთოდს, მაგალითად **login(),** და ჩვულებრივ ვაკეთებთ **POST Http**რექვესტს, ოღონდ ამ შემთხვევაში, დოკუმენტაციის მიხედვით, რექვესტს ვაგზავნით სხვა ლინკზე:

1. https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=[API\_KEY]

**Sign up**-ის მსგავსად, იგივე ტიპის body ობიექტი უნდა ჩავაწოდოთ:

1. login(email:string, password:string){
2. this.http.post('https://identitytoolkit.googleapis.com/v1/accounts:signInWithPassword?key=[API\_KEY]', {
3. email: email,
4. password: password,
5. returnSecureToken: true
6. })
7. }

ხოლო პასუხის სხეული იგივეა, უბრალოდ ერთი დამატებით: **registered (boolean)**ფროფერთი ემატება, რომელიც ამომწებს არსებობს თუ არა ექაუნთი, იგი optional-ად ჩავამატოთ უკვე არსებულ **interface**-ს.

**20. Authentication & Route Protection in Angular**

296. Sending Login Requests

**Authentication Sign in 1.2:**

**ინტერფეისი**ასე გამოიყურება შემდეგ:

1. interface AuthResponseData {
2. kind: string,
3. idToken: string,
4. email: string,
5. refreshToken: string,
6. expiresIn: string,
7. localId: string,
8. registered?:boolean
9. }

კომპონენტში კი აღნიშნულს უბრალოდ დავასუბსქრაიბებთ ზუსტად იგივეთი, როგორიც აქამდე იყო (ჩავაკოპირებთ იგივე **subscribe**-ს).

**20. Authentication & Route Protection in Angular**

296. Sending Login Requests

**Authentication Sign in 1.3:**

ან მეორე, უფრო უკეთესი გზა არის, **Observable**-ს შექმნა და მაგ ერთში გაერთიანება შემდეგნაირად:

1. let authObs: Observable<AuthResponseData>;
2. this.isLoading = true
3. if(this.isLoginMode){
4. authObs = this.authService.login(email,password)
5. }
6. else{
7. authObs = this.authService.signUp(email, password)
8. }
9. authObs.subscribe(res => {
10. console.log(res)
11. this.isLoading = false
12. }, errorMessage => {
13. console.log(errorMessage)
14. this.error = errorMessage
15. this.isLoading = false
16. })

**20. Authentication & Route Protection in Angular**

297. Login Error Handling

**Authentication Sign in  Error Handling 1.4:**

წინას მსგავსად, შეგვიძლია აღნიშნული ერორი ერთ ფუნქციაში გავაერთიანოთ და შემდეგ უკვე pipe()-ის catchError()-ს ჩავაწოდოთ აღნიშნული ფუნქცია. ასევე გავზარდოთ ერორის ტიპები. ფუნქციის კოდი შემდეგში...

**20. Authentication & Route Protection in Angular**

297. Login Error Handling

**Authentication Sign in Error Handling 1.5:**

ფუნქციის სხეული, რომელიც სხვა **pipe**-ებს ჩაეწოდებათ

1. private handleError(errRes: HttpErrorResponse){
3. let errorMessage = 'An unknown error occured!'
4. if(!errRes.error || !errRes.error.error){
5. return throwError(errorMessage);
6. }
7. switch(errRes.error.error.message){
8. case 'EMAIL\_EXISTS':
9. errorMessage = 'This email exists already';
10. break;
11. case 'EMAIL\_NOT\_FOUND':
12. errorMessage = 'This email does not exist';
13. break;
14. case 'INVALID\_PASSWORD':
15. errorMessage = 'This password is not correct';
16. break;
17. }
18. return throwError(errorMessage);
20. }

**login**და **signup**-ს, ორივეს მსგავსი pipe ექნებათ:

1. .pipe(catchError(this.handleError))

**20. Authentication & Route Protection in Angular**

298. Creating & Storing the User Data

**Authentication Create & Store user Data 1.1:**

როდესაც რექვესტს ვაგზავნით სერვერზე, აღნიშნული ასევე პასუხს გვიბრუნებს კონკრეტულ მომხმარებელზე, რომელმაც მაგალითად გაიარა რეგისტრაცია ან ავტორიზაცია. აღნიშნული ინფორმაცია მოიცავს შემდეგს:

1. **email**
2. **localId**
3. **idToken**
4. **expiresIn**

აღნიშნული ინფორმაცია უნდა შევინახოთ სადღაც.

პირველ რიგში უნდა შევქმნათ მომხმარებლის მოდელი, სადაც აღნიშნული ინფორმაცია იქნება წინასწარ დადეკლარირებული, ანუ რა ტიპის ინფორმაციას იღებს.

მას შემდეგ რაც მოდელს შევქმნით, შემდეგ შეგვიძლია სერვისშივე შევქმნათ **Subject**, რომლის ჯენერიკ ტიპიც იქნება ეს მოდელი. ხოლო აღნიშნული საბჯეცტს**.next()**-ს გაუკეთებს კონკრეტული ფუნქცია, რომელშიც შეიქმნება მომხმარებლის პროტოტიპი.

ეს ფუნქცია კი ჩაეწოდება რეგისტრაციის და ავტორიზაციის **pipe**-ებს **tap**ოპერატორში, შესაბამისი ინფორმაციის ჩართვით. კოდები შემდგომში...

**20. Authentication & Route Protection in Angular**

298. Creating & Storing the User Data

**Authentication Create & Store user Data 1.2:**

**მომხმარებლის მოდელი:**

1. export class User {
2. constructor(
3. public email: string,
4. public id: string,
5. private \_token: string,
6. private \_tokenExpirationDate: Date
7. ){}
8. get token(){
9. if(!this.\_tokenExpirationDate || new Date() > this.\_tokenExpirationDate){
10. return null;
11. }
12. return this.\_token
13. }
14. }

**get**აქ გვაძლევს საშუალებს კონკრეტულ ფროფერთის მივწვდეთ როგორც ფროფერთი და მასზე გავატაროთ გარკვეული კოდი.

**20. Authentication & Route Protection in Angular**

298. Creating & Storing the User Data

**Authentication Create & Store user Data 1.3:**

**ფუნქცია**, რომელიც ქმნის მომხმარებლის **პროტოტიპს**:

1. user = new Subject<User>()
3. private handleAuthentication(email: string, token: string, userId: string, expiresIn: number){
4. const expirationDate = new Date(new Date().getTime() + expiresIn \* 1000)
5. const user = new User(
6. email,
7. userId,
8. token,
9. expirationDate
10. )
11. this.user.next(user)
12. }

აქ ასევე პირველ რიგში ხდება **expirationDate**-ის გამოთვლა, რა დროსაც მიმდინარე დროს ემატება ის დრო, რომელსაც სერვერი გვაწვდის, რათა ვადის გასვლა დადგინდეს.

**20. Authentication & Route Protection in Angular**

298. Creating & Storing the User Data

**Authentication Create & Store user Data 1.4:**

ფუნქციის ჩაწოდება**.pipe()** მეთოდში**tap()**ოპერატორში:

1. .pipe(catchError(this.handleError),tap(resData => {
2. this.handleAuthentication(
3. resData.email,
4. resData.localId,
5. resData.idToken,
6. +resData.expiresIn
7. )
8. })
9. )

ეს მიებმება როგორც **რეგისტრაციის**, ისე **ავტორიზაციის**რექვესტებს. **+** - აღნიშნული **number**ტიპად აქცევს.

**20. Authentication & Route Protection in Angular**

299. Reflecting the Auth State in the UI

**Authentication Reflect UI:**

ამ დროს კომპონენტშ უბრალოდ **authObs**-ის დასუბსქრაიბებისას ნავიგაციას მოვახდენთ და შესაბამისად ჰედერშიც დაგვჭირდება გარკვეული მანიპულაციები იმის გასარკვევად, დაოგინებულია მომხმარებელი თუ არა, რისთვისაც ცვლადს შევქმნით, რომელიც გაარკვევს დალოგინებულები ვართ თუ არა:

**header.component:**

1. private userSub: Subscription;
2. isAuthenticated = false;
4. ngOnInit(){
5. this.userSub = this.authService.user.subscribe(user =>{
6. this.isAuthenticated = !!user;
7. console.log(!user);
8. console.log(!!user);
9. })
10. }
12. ngOnDestroy() {
13. this.userSub.unsubscribe()
14. }

ხოლო **html**-ს შესაბამისად გავუწევთ ადაპტირებას.

***\*შენიშვნა***: **!!user**- აქ **!!** იგივეა რაც ტერნერი ოპერატორი **boolean**-ის დროს

**20. Authentication & Route Protection in Angular**

300. Adding the Token to Outgoing Requests

**Authentication Adding Token 1.1:**

**Token**-ის დამატების გარეშე, ვერ გვექნება წვდომა ჩვენს **firebase**-ის სტორეიჯზე. თუმცა, მანამ ჯერ განვიხილოთ **BehaviorSubject<>(null)**, რომელიც **Subject**-ის მსგავსად შეიძლება **next**და ***subscribe***გაუკეთდეს, თუმცა მათ შორის ის განსხვავებაა, რომ **BehaviorSubject**-ისას იგი ასევე ასუბსქრაიბებს წინა მნიშვნელობას, ანუ რაც აქამდე იყო და-next-ებული.

შემდეგ თავად სტორეიჯის სერვისში გადავდივართ და აღნიშნულ user-ს ვიძახებთ, რომელიც დაკავშირებულია **BehaviorSubject-**თან. ვინაიდან აღნიშნული მომხმარებელი მხოლოდ ერთხელ გვჭირდება, შეგვიძლია ეგრევე დავაანსუბსქრაიბოთ (this.authService.user.subscribe().unsubscribe()), ან უბრალოდ გამოვიძახოთ pipe და ჩავაწოდოთ ახალი ოპერატორი **take()**, რომელიც არგუმენტად იღებს რიცხვს, რომლითაც განისაზღვრება რა რაოდენობის ინფორმაცია/მნიშნველობა გსურს კონკრეტული სუბიექტიდან და იგი მას შემდეგ ავტომატურად დააანსუბსქრაიბებს:

this.authService.user.pipe(take(1)).subscribe()

**20. Authentication & Route Protection in Angular**

300. Adding the Token to Outgoing Requests

**Authentication Adding Token 1.2:**

შემეგ თვითონ **subscribe**-ში შეგვეძლო გამოგვეყენებინა **Http**რექვესტი (რომელიც მის ქვედა ფუნქციაში **return**-დება (**get**)) და მიგვემაგრებინა **Token**, თუმცა **Observable**-ში ვერ შევძლებთ **return**-ის გაკეთებას.

ამის გამოსავალი არის შემდეგში: **pipe()** უნდა გავავრცელოთ ორივე **Observable**-ზე და დავა-**return**-ოთ ერთი დიდი **Observable**.

ამისთვის გამოიყენება **rxjs**-ის ოპერატორი: **exhaustMap().**

**exhaustMap**იგი ელოდება პირველ **Observable**-ს რომ დაასრულოს მუშაობა, შემდეგ კი აღნიშნულ **Observable**-ს აწვდის მეორე **Observable**-ს და პირველს აწერს ზევიდან, ანუ საბოლოო ჯამში მეორე  **Observable**-ს დააბრუნებს, რომელსაც პირველის გავლენა ექნება.

შემდეგ კი get რექვესტს ჩავსვამთ **exhaustMap**-ში, ხოლო მას გავაგრძელებთ დარჩენილი rxjs ოპერატორებით, ანუ უბრალოდ გადავაბამთ.

**20. Authentication & Route Protection in Angular**

300. Adding the Token to Outgoing Requests

**Authentication Adding Token 1.3:**

გამოცალკევებული რექვესტისას, რომელიც return-ის გამო არ იმუშავებს:

1. fetchRecipes(){
2. this.authService.user.pipe(take(1)).subscribe(
3. user => {
5. }
6. )
7. return this.http.get<Recipe[]>('https://ng-course-recipe-book-3d899-default-rtdb.firebaseio.com/recipes.json').
8. pipe(map(recipes => {
9. return recipes.map(recipe => {
10. return {...recipe, ingredients: recipe.ingredients ? recipe.ingredients : []}
11. })
12. }),
13. tap(recipes => {
14. this.recipeService.setRecipes(recipes)
15. })
16. )
17. }

**20. Authentication & Route Protection in Angular**

300. Adding the Token to Outgoing Requests

**Authentication Adding Token 1.4:**

ერთ **.pipe()**-ში გაერთიანებისას:

1. fetchRecipes(){
3. return this.authService.user.pipe(
5. take(1)
7. ,exhaustMap(user=>{
8. return this.http.get<Recipe[]>('https://ng-course-recipe-book-3d899-default-rtdb.firebaseio.com/recipes.json');
9. })
11. ,map(recipes => {
12. return recipes.map(recipe => {
13. return {...recipe, ingredients: recipe.ingredients ? recipe.ingredients : []}
14. })
15. })
17. ,tap(recipes => {
18. this.recipeService.setRecipes(recipes)
19. })
21. );
23. }

**20. Authentication & Route Protection in Angular**

300. Adding the Token to Outgoing Requests

**Authentication Adding Token 1.5:**

იმისათვის, რომ აღნიშნული ავამუშავოთ, **firebase**-ის შემთხვევაში **query**პარამეტრი უნდა ჩავამატოთ, სხვა სერვერის შემთხვევაში უბრალოდ ჰედერები ემატება.

შემდეგ უკვე ვამატებთ ჩვენს **token**-ს, ამისთვის ან ხელით ჩავუწეროთ თავად ლინკს:

'https://ng-course-recipe-book-3d899-default-rtdb.firebaseio.com/recipes.json?auth='+ user.token

get რექვესტს მეორე არგუმენტად უნდა ჩავაწოდოთ ობიექტი, რომელშიც იქნება ფროფერთ **params**და იგი მნიშნველობად მიიღებს:

new HttpParams().set('auth', user.token)

**კოდში:**

1. exhaustMap(user=>{
2. return this.http.get<Recipe[]>('https://ng-course-recipe-book-3d899-default-rtdb.firebaseio.com/recipes.json', {
3. params: new HttpParams().set('auth', user.token)
4. });
5. })

**20. Authentication & Route Protection in Angular**

301. Attaching the Token with an Interceptor

**Authentication Token Interceptor 1.1:**

ჩვენ შეგვეძლო წინა მოქმედება, რომელიც მხოლოდ **fetchRecipe**-ს შეეხებოდა უბრალოდ გაგვემეორებინა **storeRecipes**-შიც, თუმცა ამის ნაცვლად გამოვიყენებთ **Interceptor**-ს.

**exhaustMap**-ის ლოგიკა გადაგვაქვს **Interceptor**-ის სერვისში, რომელსაც ცალკე ვქმნით.

კოდები შემდეგში...

**20. Authentication & Route Protection in Angular**

301. Attaching the Token with an Interceptor

**Authentication Token Interceptor 1.2:**

**Interceptor:**

1. constructor(private authService: AuthService){}
3. intercept(req: HttpRequest<any>, next: HttpHandler){
4. return this.authService.user.pipe(
6. take(1)
8. ,exhaustMap(user => {
9. if(!user){
10. return next.handle(req)
11. }
12. const modifiedReq = req.clone({
13. params: new HttpParams().set('auth', user.token)
14. })
15. return next.handle(modifiedReq)
16. })
18. )
20. }

**20. Authentication & Route Protection in Angular**

301. Attaching the Token with an Interceptor

**Authentication Token Interceptor 1.3:**

**data-storage.service (იგივე რჩება თითმქმის):**

1. fetchRecipes(){
2. return this.http.get<Recipe[]>('https://ng-course-recipe-book-3d899-default-rtdb.firebaseio.com/recipes.json')
4. .pipe(
6. map(recipes => {
7. return recipes.map(recipe => {
8. return {...recipe, ingredients: recipe.ingredients ? recipe.ingredients : []}
9. })
10. })
12. ,tap(recipes => {
13. this.recipeService.setRecipes(recipes)
14. })
16. )
18. }

**20. Authentication & Route Protection in Angular**

302. Adding Logout

**Authentication LogOut button:**

ამისთვის უბრალოდ არსებულ **BehaviorSubject**-ის **next**-ს ჩავაწვდით **null**-ს რომელიც გაიწერება ახალ მეთოდში, რომელიც შეიქმნება აუთენტიკაციის სერვისში და შემდეგ დაკავშირდება ჰედერის კომპონენტის მეთოდთან. ასევე აღნიშნულ სერვისის მეთოდში წავა ნავიგაციის განმსაზღვრელი:

1. logout() {
2. this.user.next(null);
3. this.router.navigate(['/auth']);
4. }

**20. Authentication & Route Protection in Angular**

303. Adding Auto-Login

**Authentication auto-login:**

ამისთვის აუთენტიკაციის სერვისშ შევქმნით ახალ მეთოდს, რომელსაც შემდეგ **app.component**-ის ლოგიკაში **OnInit**-ში გამოვიძახებთ. ვიყენებთ **localStorage**-ს:

1. autoLogin(){
2. const userData: {
3. email: string;
4. id: string;
5. \_token: string;
6. \_tokenExpirationDate: string;
7. } = JSON.parse(localStorage.getItem('userData'))
9. if(!userData){
10. return;
11. }
13. const loadedUser = new User(userData.email, userData.id, userData.\_token, new Date(userData.\_tokenExpirationDate))
15. if(loadedUser.token){
16. this.user.next(loadedUser)
17. }
18. }

**20. Authentication & Route Protection in Angular**

304. Adding Auto-Logout

**Authentication auto-logout 1.1:**

ამისთვის უნდა შევქმნათ აუთენტიკაციის სერვისში ცალკე მეთოდი და შემდეგ აღნიშნული მეთოდი ჩავაწოდოთ რამდენიმე სხვა მეთოდში. თავად მეთოდი:

1. autoLogout(expirationDuration: number){
2. this.tokenExpirationTimer = setTimeout(()=>{
3. this.logout()
4. },expirationDuration)
5. }

**handleAuthentication**-ში:

1. private handleAuthentication(
2. email: string,
3. userId: string,
4. token: string,
5. expiresIn: number
6. ) {
7. const expirationDate = new Date(new Date().getTime() + expiresIn \* 1000);
8. const user = new User(email, userId, token, expirationDate);
9. this.user.next(user);
10. this.autoLogout(expiresIn \* 1000);
11. localStorage.setItem('userData', JSON.stringify(user));
12. }

**20. Authentication & Route Protection in Angular**

304. Adding Auto-Logout

**Authentication auto-logout 1.2:**

**autoLogin**-ში **token**ქონდიშენის შემოწმებისას:

1. if(loadedUser.token){
2. this.user.next(loadedUser);
3. const expirationDuration = new Date(userData.\_tokenExpirationDate).getTime() - new Date().getTime();
4. this.autoLogout(expirationDuration)
5. }

ასევე მნიშვნელოვანია აღნიშნულის გაწმენდა როცა **logout**მოხდება, ეს კი განხორციელდება **logout()** მეთოდში:

1. logout() {
2. this.user.next(null);
3. this.router.navigate(['/auth']);
4. localStorage.removeItem('userData');
5. if(this.tokenExpirationTimer){
6. clearTimeout(this.tokenExpirationTimer)
7. }
8. this.tokenExpirationTimer = null;
9. }

**20. Authentication & Route Protection in Angular**

305. Adding an Auth Guard

**Authentication AuthGuard 1.1:**

როუტინგიზე გარდის დასაყენებლად ცალკე სერვისი უნდა შევქმნათ, რომელიც იმუშავებს **CanActivate**-ით. აღნიშნული სერვისის იდეალური ვარიანტი:

1. @Injectable({providedIn: 'root'})
3. export class AuthGuard implements CanActivate {
5. constructor(private authService: AuthService, private router: Router){}
7. canActivate(
8. route: ActivatedRouteSnapshot,
9. router: RouterStateSnapshot): boolean | Promise<boolean | UrlTree> | Observable<boolean | UrlTree>
10. {
11. return this.authService.user.pipe(
12. take(1),
13. map(user => {
14. const isAuth = !!user
15. if(isAuth){
16. return true
17. }
18. return this.router.createUrlTree(['/auth'])
19. }),
20. );
21. }
22. }

**20. Authentication & Route Protection in Angular**

305. Adding an Auth Guard

**Authentication AuthGuard 1.2:**

***თუმცა:***

შეგვეძლო შემდეგნაირადაც გვექნა: **UrlTree**-ის ნაცვლად, როგორც ადრე, პირდაპირ **tap()** ოპერატორით როუტინგი შეგვეცვალა:

1. @Injectable({providedIn: 'root'})
3. export class AuthGuard implements CanActivate {
5. constructor(private authService: AuthService, private router: Router){}
7. canActivate(
8. route: ActivatedRouteSnapshot,
9. router: RouterStateSnapshot): boolean | Promise<boolean> | Observable<boolean>
10. {
11. return this.authService.user.pipe(map(user => {
12. return !!user
13. }), tap(isauth =>{
14. if(!isauth){
15. this.router.navigate(['/auth'])
16. }
17. }));
18. }
19. }

**21. Dynamic Components**

312. Preparing Programmatic Creation

**Dynamic Components Creating 1.1:**

კომპონენტების დინამიურად გამოჩენა შეგვიძლია **\*ngIf**-ის საშუალებით, თუმცა მათი შექმნის სხვა გზაც არსებობს. ეს გზა პირდაპირ ლოგიკაში გაწერაზე გადის.

პირველ რიგში ეს კოდი უნდა გავწეროთ იმ კომპონენტის ლოგიკაში, სადაც გვსურს გამოძახება.

ამისთვის ჯერ უნდა დავაინჯექტოთ **constructor** **ComponentFactoryResolver**-ით, რომელიც იმპორტდება **@angular/core**-დან.

შემდეგ კი კონკრეტულ private მეთოდში ცვლადში შევინახოთ აღნიშნული ინჯექტი, სადაც მის მეთოდს - **resolveComponentFactory()** მივაბამთ. ამას კი არგუმენტად ჩაეწოდება ის კომპონენტი, რომელიც გვსურს რომ დინამიურად გამოვაჩინოთ ან გავაქროთ კოდიდან.

1. private showErrorAlert(message: string){
2. const alertCompFactory = this.componentFactoryResolver.resolveComponentFactory(AlertComponent);
3. }

შემდეგ კი აღნიშნული კომპონენტის **DOM**-ის ლოკაციის განსასაზღვრად საჭიროა შევქმნათ დირექტივი. აღნიშნული დირექტივი დაინჯექტდება **ViewContainerRef**-ით.

**21. Dynamic Components**

312. Preparing Programmatic Creation

**Dynamic Components Creating 1.2:**

**ViewContainerRef**საშუალებას გაძლევს მიმაგრების ელემენტზე არა მხოლოდ მიიღო წვდომა, ანუ მის ადგილმდებარეობაზე, არამედ გამოიყენო ისეთი მეთოდი, რომელიც იმ ადგილზე ახალ ელემენტს შექმნის.

**Directive:**

1. import { Directive, ViewContainerRef } from "@angular/core";
3. @Directive({
4. selector: '[appPlaceholder]'
5. })
7. export class PlaceholderDirective {
9. constructor(public viewContainerRef: ViewContainerRef){}
11. }

**21. Dynamic Components**

313. Creating a Component Programmatically

**Dynamic Components Creating 1.3:**

შემდეგ ეტაპზე თემფლეითში (**HTML**) ვქმნით **ng-template**-ს, რომელიც თავისთავად არაფერს არ არენდერებს **DOM**, თუმცა მაინც ხელმისაწვდომია ანგულარის თემფლეითინგის ენაში.

შემდეგ აღნიშნულ კომპონენტს ლოგიკიდან შეგვიძლია მივწვდეთ **@ViewChild()**-ის გამოყენებით, რომელსაც არგუმენტად შეგვიძლია ჩავაწოდოთ არა ლოქალრეფერენსი, არამედ თავად ტიპი, ანუ ის დირექტივი, რომელიც მასზეა მიმაგრებული (არა მიმაგრების სახელით, არამედ თავად დირექტივის კლასის, რომელიც ექსპორტდება და აქ უნდა დაიმპორტდეს) და ფროფერთიში, რომელსაც მას შევინახავთ, იგივე დირექტივის ტიპი ექნება:

1. @ViewChild(PlaceholderDirective, {static: false}) alertHost: PlaceholderDirective;

**21. Dynamic Components**

313. Creating a Component Programmatically

**Dynamic Components Creating 1.4:**

შემდეგ აღნიშნული ფროფერთი შეგვიძლია შემქმნელ მეთოდში გამოვიძახოთ, ანუ შევინახოთ ცვლადში და მივწვდეთ მის **viewContainerRef**-ს, რამდენადაც აღნიშნული დირექტივში **public**-ზე აყენია:

1. private showErrorAlert(message: string){
2. const alertCompFactory = this.componentFactoryResolver.resolveComponentFactory(AlertComponent);
3. const hostViewContainerRef = this.alertHost.viewContainerRef
4. hostViewContainerRef.clear()
5. hostViewContainerRef.createComponent(alertCompFactory)
6. }

აქედან **clear()** მეთოდი გაწმენდს ყველა სხვა კონტენტს, რომელიც აღნიშნულ ადგილზე აქამდე დარენდერებულა.

მას შემდეგ უკვე შეგვიძლია გამოვიყენოთ **createComponent()** მეთოდი, რომელსაც არგუმენტად ჩაეწოდება **alertCompFactory**(*სადაც ვიღებთ წვდომას კომპონენტზე*).

*თუმცა****error****-ს მოგვცემს, რასაც სამომავლოდ გამოვასწორებთ!*

**21. Dynamic Components**

314. Understanding entryComponents

**Dynamic Components Creating 1.5:**

წინაზე ნახსენებ **error**-ს მოგვცემდა, თუ **Angular 9**-ზე დაბალ ვერსიას ვიყენებთ, რადგან ანგულარი შემდეგნაირად მუშაობს:

ავტომატურად მას კომპონენტის არსებობაზე შეუძლია შეამოწმოს ინფორმაცია თავად თემფლეითიდან, სადაც კომპონენტია გამოძახებული ან როუტინგიდან, ხოლო როცა ხელით ვქმნით, მას ამ კომპონენტის შესამოწმებლად ჭირდება **appModule**-ში **bootstrap**-ის ქვემოთ ახალი ფროფერთის დამატება, რომელიც ასევე მასივია, რომელსაც ეწოდება: **entryComponents**, რომელშიც წავა იგივე კომპონენტი:

1. entryComponents: [
2. AlertComponent
3. ]

**21. Dynamic Components**

315. Data Binding & Event Binding

**Dynamic Components Creating (Data & Event binding) 1.6:**

იმისათვის, რომ კომპონენტს ინფორმაცია ჩავაწოდოთ, აღნიშნული კომპონენტის შექმნის ბრძანება უნდა შევინახოთ ცვლადში და შემდგომ ამ ცვლადს მივაბათ ფროფერთი - **instance**, რომელშიც იქნება ინფორმაცია იმ ფროფერთიებზე, რომლებიც იმყოფებიან ჩვენს კომპონენტში:

1. componentRef.instance.message = message

ხოლო მეთოდებზე წვდომისთვის, განსხვავებული მიდგომაა. რამდენადაც ჩვენს კომპონენტში **@Output** დეკორატორი გამოვიყენეთ, მისი გარედან დასუბსქრაიბება ჩვეულებრივ არ შეიძლება ხოლმე, თუმცა ამ შემთხვევაში და მხოლოდ ამ შემთხვევაშია გამონაკლისი, როცა ეს შეგვიძლია. თუმცა აღნიშნული უნდა შევინახოთ ფროფერთიში, რომელსაც შემდეგ დავაანსუბსქრაიბებთ, ხოლო მომდევნო იქნება **clear()** მეთოდის გამოყენება:

1. private closeSub: Subscription
3. componentRef.instance.message = message
4. this.closeSub =  componentRef.instance.close.subscribe(()=> {
5. this.closeSub.unsubscribe();
6. hostViewContainerRef.clear()
7. })

**21. Dynamic Components**

315. Data Binding & Event Binding

**Dynamic Components Creating (Data & Event binding) 1.7:**

მთლიანი მეთოდის კოდი მისი დამხმარე ფროფერთიებით:

1. @ViewChild(PlaceholderDirective, {static: false}) alertHost: PlaceholderDirective;
2. private closeSub: Subscription

5. private showErrorAlert(message: string){
6. const alertCompFactory = this.componentFactoryResolver.resolveComponentFactory(AlertComponent);
7. const hostViewContainerRef = this.alertHost.viewContainerRef;
9. hostViewContainerRef.clear()
11. const componentRef = hostViewContainerRef.createComponent(alertCompFactory)
13. componentRef.instance.message = message
14. this.closeSub =  componentRef.instance.close.subscribe(()=> {
15. this.closeSub.unsubscribe();
16. hostViewContainerRef.clear()
17. })
18. }

**21. Dynamic Components**

315. Data Binding & Event Binding

**Dynamic Components Creating (Data & Event binding) 1.8:**

დაზღვევის მიზნით, ასევე სასურველია, თუ გამოძახებული მეთოდი ისევ დასუბსქრაიბებულია, **ngOnDestroy**-ში დავაანსუბსქრაიბოთ:

1. ngOnDestroy(){
2. if(this.closeSub){
3. this.closeSub.unsubscribe()
4. }
5. }

**22. Angular Modules & Optimizing Angular Apps**

319. What are Modules?

**Modules Introduction:**

მოდულები გამოიყენება ანგულარის კომპონენტების, სერვისების, პაიპების, დირექტივების და ა.შ. ერთად შესაკრავად, ერთ ძირითად მოდულში.

**22. Angular Modules & Optimizing Angular Apps**

320. Analyzing the AppModule

**Modules appModule:**

* **Declarations**: დეკლარირდება ის კომპონენტები, დირექტივები, ქოსტუმ პაიპები და ა.შ., რომლებიც ჩვენი აპლიკაციის ნაწილებია;
* **imports**: აიმპორტებს ანგულარში ჩაშენებულ სხვა მოდულებს
* **providers**: აქ სერვისები მიდის, რადგან მათზე წვდომა ყველა კომპონენტს ჰქონდეს.
* **bootstrap**: ხედავს იმ კომპონენტს, რომელიც გამოძახებულია *index.html*ფაილში (აქ შეიძლება იყოს სხვა მრავალი კომპონენტიც).
* **entryComponents**: გამოიყენება ლოგიკაში შექმნილი კომპონენტების დასადეკლარირებლად

მოდულები ერთმანეთთან ავტომატურად არ ამყარებენ კომუნიკაციას, ანუ დამოუკიდებლად მუშაობენ, ამისთვის გამოიყენება **export**ფროფერთი, რომელიც სხვა მოდულში **import**-ს გაუკეთებს კონკრეტული მოდულის კონტენტს.

**22. Angular Modules & Optimizing Angular Apps**

321. Getting Started with Feature Modules

**Modules MutliModule Intro 1.1:**

მოდულის გახლეჩვა რამდენიმე მოდულად პირველ რიგში სასარგებლოა იმ შემთხვევაში, თუ დიდი აპლიკაციაა და პერფორმანსის გაზრდა გვსურს.

მაგალითად, რეცეპტებს შევუქმენით ცალკე მოდული, აღნიშნული შემდეგნაირად გამოიყურება:

1. import { NgModule } from "@angular/core";
2. import { RecipesComponent } from "./recipes.component";
4. @NgModule({
5. declarations:[
6. RecipesComponent,
7. ],
8. exports:[
9. RecipesComponent,
10. ]
11. })
12. export class RecipesModule {}

ხოლო **appModule**-ში ამ კონკრეტულ მოდულს ვაიმპორტებთ:

1. imports: [
2. RecipesModule
3. ],

ამის შემდეგ, ეს კონკრეტული კომპონენტი ხელმისაწვდომი იქნება **appModule**-თან დაკავშირებულ ყველა კომპონენტში, თუმცა, არა პირიქით, ანუ **appModule**-ში დაიმპორტებული მოდულები ვერ იქნება კავშირში ჩვენს **RecipesModule**-თან (სერვისები მაინც ხელმისაწვდომი იქნება). *ამისთვის:*

**22. Angular Modules & Optimizing Angular Apps**

322. Splitting Modules Correctly

**Modules MutliModule Intro 1.2:**

ამის მოსაგვარებლად შეგვიძლია ჩვენს მიერ შექმნილ მოდულში დავაიმპორტოთ **AppRoutingModule**- როუტინგისთვის, ხოლო ანგულარის ბეისიკ **feature**-ებისთვის - **CommonModule**.

**appModule**ამ ქომონ ფიჩურეფისთვის იყენებს **BrowserModule**, თუმცა მანდ ასევე ისეთი ფიჩურებიც შედის, რაც მხოლოდ ერთხელ უნდა დარენდერდეს, ამიტომ უკეთესია **CommonModule**-ის გამოყენება ***(common-ში ასევე შედის მაგალითად \*ngIf და \*ngFor)***:

1. import { NgModule } from "@angular/core";
2. import { CommonModule } from "@angular/common";
3. import { RecipesComponent } from "./recipes.component";
4. import { ReactiveFormsModule } from "@angular/forms";
6. @NgModule({
7. declarations:[
8. RecipesComponent,
9. ],
10. imports:[AppRoutingModule, CommonModule, ReactiveFormsModule],
11. providers:[],
12. exports:[
13. RecipesComponent,
14. ]
15. })
17. export class RecipesModule {}

**22. Angular Modules & Optimizing Angular Apps**

323. Adding Routes to Feature Modules

**Modules Routing Movement:**

გარდა იმ ყველაფრისა, რაც გავაკეთეთ, ასევე შეგვიძლია როუტინგის კუთვნილი როუტის გადატანა ჩვენს მიერ შექმნილ მოდულში. ამისთვის:

შეგვიძლია იგივე მოდულში ავაწყოთ, ან ცალკე როუტინგ მოდული შევქმნათ იგივე პრინციპით, უბრალოდ ერთი განსხვავებით, **forRoot()**-ის ნაცვლად გამოვიყენებთ **forChild()**-ს:

1. @NgModule({
2. imports:[RouterModule.forChild(routes)],
3. exports:[RouterModule]
4. })

**22. Angular Modules & Optimizing Angular Apps**

324. Component Declarations

**Modules exp Comp:**

ასევე მნიშვნელოვანია ვიცოდეთ, რომ როუტინგი არ იმუშავებს, თუ **declarations**-ში არ გვაქვს შემოტანილი კონკრეტული კომპონენტი.

ასევე, რამდენადაც შიდა როუტინგი გვაქვს, აღარაა საჭირო დაექსპორტება კომპონენტების, რადგან ისინი სხვაგან დამოუკიდებლად არ გამოიძახებიან, ამიტომ შეგვიძლია წავშალოთ:

1. import { CommonModule } from "@angular/common";
2. import { NgModule } from "@angular/core";
3. import { ReactiveFormsModule } from "@angular/forms";
4. import { AppRoutingModule } from "../app-routing.module";
5. import { RecipesRoutingModule } from "./recipes-routing.module";
6. import { RecipesComponent } from "./recipes.component";

9. @NgModule({
10. declarations:[
11. RecipesComponent,
12. ],
13. imports:[AppRoutingModule, CommonModule, ReactiveFormsModule, RecipesRoutingModule],
14. providers:[]
15. })
16. export class RecipesModule {}

**22. Angular Modules & Optimizing Angular Apps**

326. Understanding Shared Modules

**Modules Shared Modules:**

აქამდე ვიყენებდით **Feature Module**-ს. ახლა გადავდივართ **Shared Modules**-ზე:

ამისთვის უბრალოდ შევქმნით ცალკე მოდულს, რომელიც შეიართებს საზიარო კომპონენტებს და ა.შ. და შემდეგ დააექსპორტებს.

სხვა მოდულები ცალ-ცალკე დაიიმპორტებენ ამ მოდულს და ისე მიწვდებიან სასურველ კომპიონენტს ან ა.შ.

\*შენიშვნა: **declaration**-ში ჩაწერილი რაიმე მხოლოდ ერთხელ შეიძლება იქნას გამოყენებული, **imports**რამდენჯერაც გვინდა. ამიტომ, ერთხელ უნდა დადეკლარირდეს რომელიმე მოდულში და შემდეგ უბრალოდ დაექსპორტდეს.

**22. Angular Modules & Optimizing Angular Apps**

327. Understanding the Core Module

**Modules Core Module (სერვისები):**

აღნიშნულით სარგებლობა შეგვიძლია, თუ სერვისში providedIn: 'root'-ს არ ვიყენებთ, ანუ სერვისებს **appModule**-ის provedrs-ის მასივში ვადეკლარირებთ.

ამისთვის შეგვიძლია ცალკე შევქმნათ app ფოლდერში**core.module.ts** სადაც სტანდარტულ მოდულს შევქმნით და შემდეგ უკევ **appModule**-ში უბრალოდ დავაიმპორტებთ **CoreModule**-ს:

**CoreModule:**

1. @NgModule({
2. providers:[
3. ShoppingListService,
4. RecipeService,
5. {
6. provide: HTTP\_INTERCEPTORS,
7. useClass: AuthInterceptorService,
8. multi: true
9. }
10. ]
11. })
13. export class CoreModule{}

**appModule:**

1. imports: [
2. CoreModule
3. ],

**22. Angular Modules & Optimizing Angular Apps**

329. Understanding Lazy Loading

**Modules Lazy Loading:**

იგი მოითხოვს სხვადასხვა მოდულის შექმნას, რათა კონკრეტულ მომენტში კონკრეტული კონტენტი ჩატვირთოს და არა მთლიანად.

ეს პირდაპირ დიდ გავლენას ახდენს პერფორმანსზე.

**22. Angular Modules & Optimizing Angular Apps**

330. Implementing Lazy Loading

**Modules Lazy Loading implementing 1.1:**

იგი ასევე მოითხოვს, რომ როუტინგის მოდული ცალკე იყოს გამოყვანილი და ასევე **RouterModule**იყენებდეს **forChild()** მეთოდს.

ამისთვის დამოუკიდებელ როუტინგში, მაგალითად რეცეპტების როუტინგში, ძირითად **path**-ს ვუწერთ ცარიელ სტრინგს: **path: ''**, შემდეგ კი მთავარ როუტინგში ვამატებთ ახალ როუტის ობიექტს და იქ ვწერთ ამ კონკრეტულ **path**-ს:

1. { path: 'recipes', loadChildren: './recipes/recipes.module.ts#RecipesModule'}

**loadChildren**განსაზღვრავს, რომ აღნიშნულ როუტზე მდგარი მოდული მაშინ ჩაიტვირთოს, როცა **recipes**-ს ლინკი გააქტიურდება.

იგი წვდება თავად მოდულს ამ კონკრეტულის, ანუ არა როუტს, არამედ მოდულს და **#**-ის მერე წერს კონკრეტული მოდულის კლასის სახელს.

***\*შენიშვნა:*** ყურადღებით უნდა ვიყოთ აღნიშნულ მოდულში იმპორტებზე, რადგან ისინი ერთ კონკრეტულ ბანდლში ერთიანდებიან. ასევე სასურველია lazyLoading-ის დამატების მერე თავიდან დავბილდოთ.

**22. Angular Modules & Optimizing Angular Apps**

330. Implementing Lazy Loading

**Modules Lazy Loading implementing 1.2:**

ანგულარის ახალი ვერსიებისთვის სინტაქსი მცირედით იცვლება:

1. loadChildren: () => import('./recipes/recipes.module').then(m => m.RecipesModule)}

ასვე მნიშვნელოვანია, რომ **appModule**-ში კონკრეტული მოდული **იმპორტებიდან ამოვიღოთ**, რათა არ დარენდერდეს.

**22. Angular Modules & Optimizing Angular Apps**

332. Preloading Lazy-Loaded Code

**Modules Lazy Loading Preloading:**

იგი არაფერს არ ცვლის, გარდა იმისა, რომ შემდგომში ჩამოსატვირთი ფაილების ჩამოტვირთვას მთავარი ფაილის ჩამოტვირთვის მერე ეგრევე იწყებს, რათა მომხმარებელი რომ გამოიძახებს კონკრეტულ მოდულს, უკვე ჩამოტვირთული იყოს და ლოდინი არ მოუწიოს.

მის მისაღწევად მთავარ **app-routing.module**-ში **forRoot()**-მეთოდს მეორე არგუმენტად უნდა ჩავაწოდოთ ობიექტი შემდეგნაირად:

1. @NgModule({
2. imports: [RouterModule.forRoot(appRoutes, {preloadingStrategy: PreloadAllModules})],
3. exports: [RouterModule]
4. })

*\*შენიშვნა: კონკრეტული ობიექტის უფრო მეტი კონფიგურირებაც შეიძლება*

**22. Angular Modules & Optimizing Angular Apps**

333. Modules & Services

**Modules Lazy-Loaded Module (Services):**

თუ ზარმაც ჩამტვირთველ მოდულში providers-ში გავწერთ სერვისს, იგი მაშინ იქნება ხელმისაწვდომი, როცა კონკრეტული მოდული ჩაიტვირთება და არა მანამდე.

**22. Angular Modules & Optimizing Angular Apps**

335. Ahead-of-Time Compilation

**Modules Compiling:**

**ng serve** - იყენებს **Just-in-Time (JiT)** კომპაილერს, რადგან იგი საუკეთესოა დებაგინგისთვის.

**ng build --prod** - ეს **CLI** ბრძანება ქმნის აპლიკაციის ბანდლებს ერთ რამდენიმე პატარა მოცულობის ფაილში (**Ahead-of-Time Compilation**).

ბილდინგისას შეიძლება ერთ-ერთმა**\*ngFor**-მა წარმოქმნას ერორი, რომელიც რეაქტიული ფორმსის ფორმების მასივის კომპონენტის ნაწილში გვიწერია. ამის მოსაგვარებლად, იგი **ts**-ში შეიძლება გადავიდეს:

1. get ingredientsControls() {
2. return (this.recipeForm.get('ingredients') as FormArray).controls
3. }

**html**-ში კი:

1. \*ngFor="let ingredientCtrl of ingredientsControls; let i = index"

**23. Deploying an Angular App**

340. Using Environment Variables

**Deployment Environment Variables:**

თუ აპლიკაციაში გამოყენებული გვაქვს რაიმე **API**, მაშინ სასურველია მათ **key**-ები შევინახოთ **environments**-ის ფოლდერში, **environment.ts** და **environment.prod.ts***(ბილდის მერე გამოჩნდება)*ფაილში შემდეგნაირად:

1. export const environment = {
2. production: true,
3. firebaseAPIKey: 'key აქ წავა'
4. };

შემდეგ კი კონკრეტულ სერვისში ან კომპონენტში, სადაც ეგ **key**არის გამოყენებული, უბრალოდ დავაიმპორტებთ ამ ფაილს და შემდეგნაირად დავაკავშირებთ:

1. import { environment } from '../../environments/environment';
3. ...
4. 'https://www.googleapis.com/identitytoolkit/v3/relyingparty/signupNewUser?key=' + environment.firebaseAPIKey
5. ..

**23. Deploying an Angular App**

341. Deployment Example: Firebase Hosting

**Deployment Deploying via FireBase:**

ამისთვის პირველ რიგში გვჭირდება **firebase CLI:**

npm install -g firebase-tools

შემდეგ უნდა უნდა გავუშვათ ბრძანება:

firebase login

შემდეგი ბრძანებაა (ამის გასაშვებად უკვე უნდა ვიყოთ ჩვენი პროექტის ფილდერში, ანუ უკვე **დაბილდულში**):

firebase init

შემდეგ უნდა ავირჩიოთ რომელი სერვისის გამოყენება გვსურს, აქედან ვირჩევთ მხოლოდ **Hosting**-ს (**space**-ზე დაჭერით), შემდეგ უნდა ავირჩიოთ: მიმდინარე პროექტი (**use an existing project**)

შემდეგ უნდა მივუთითოთ რომელ ფოლდერში გვაქვს ჩვენი დაბილდული აპლიკაცია: **dist/ng-complete-guide-update**

შემდეგ ვირჩევთ **single-page app**-ში **y**-ს (რადგან **index.html**-ს ემსახუროს ყველა რექვესტი)

შემდეგ **github**-ზე - **n**

ბოლო ბრძანებაა: firebase deploy

ბოლოს მოგვცემს hosting URL-ს, სადაც შეგვიძლია ჩვენი აპლიკაციაა