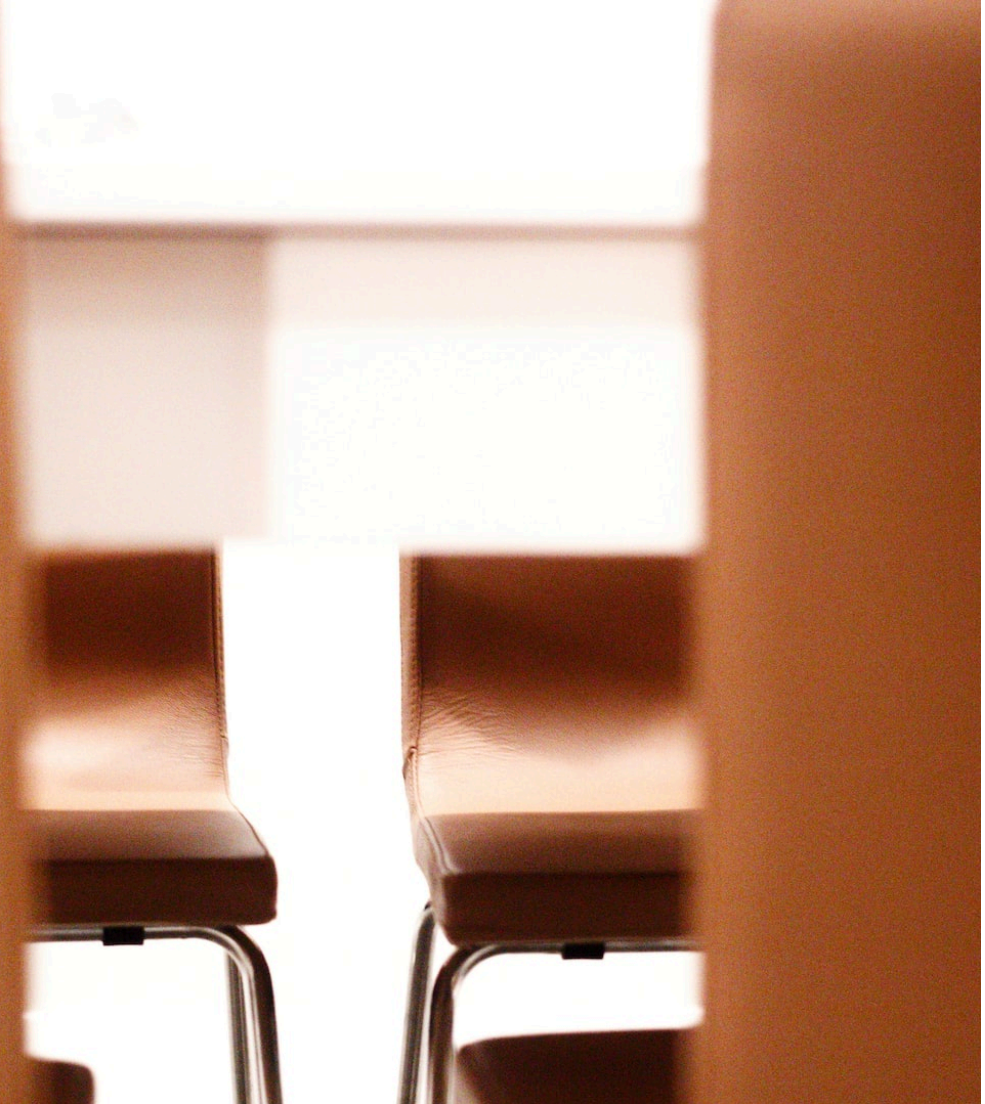
Effective creation of web applications with

Angular



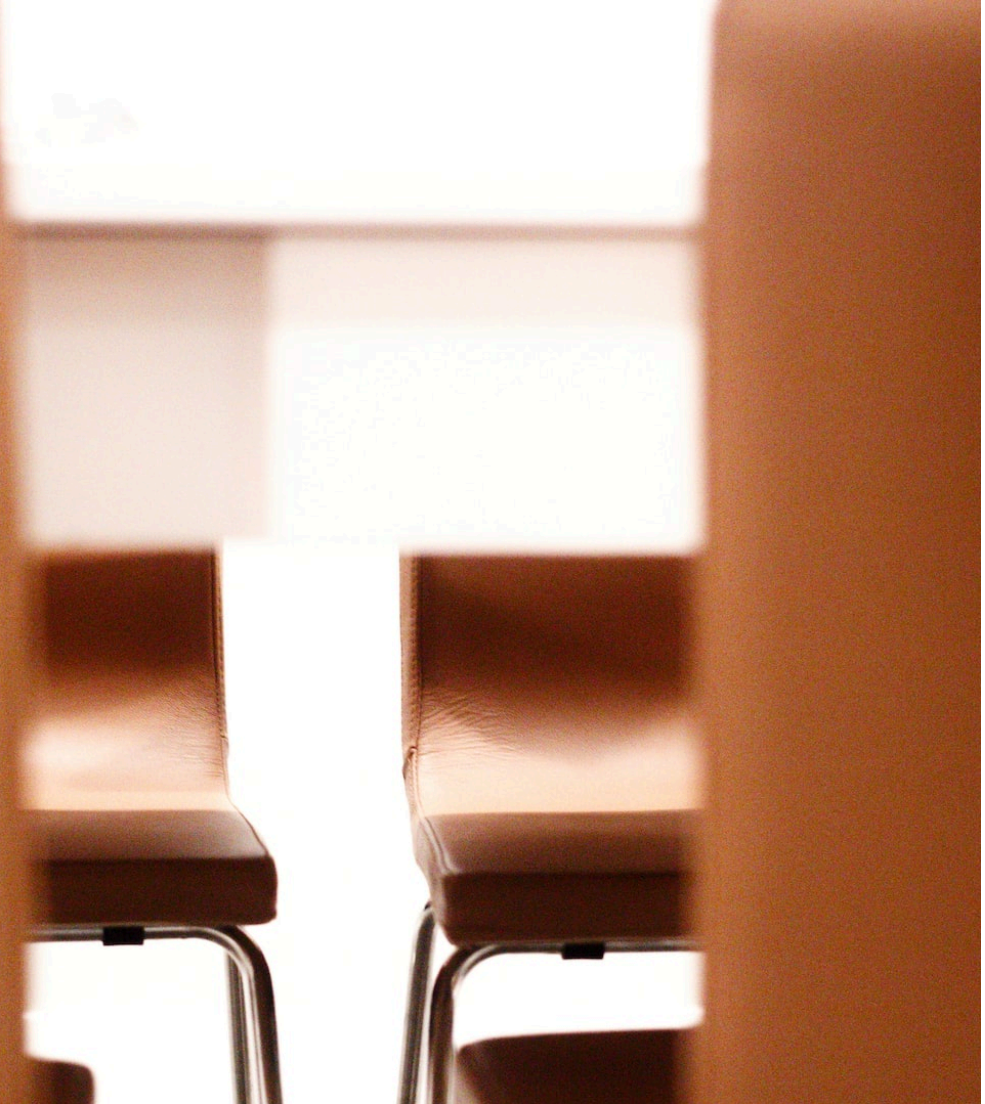
Training

- training plan and goals
- current duties
- questions, discussions, group needs
- active participation
- flexible training program
- feel free to ask questions

Mateusz Kulesza

The trainer

- Senior Software Developer
- Team Leader
- Project Manager
- Consultant and Trainer



You

the participant

Few Quick Questions:

- Your daily responsibilities
- Technologies and background
- Previous Exprience
- Your goals for this training
- Specifics areas or topics

Required software

Installation

https://nodejs.org/en

```
node -v
npm -v
```

GIT

https://git-scm.com/download/win

git -v

Alternatively NVM

https://github.com/coreybutler/nvm-windows

```
nvm install lts
nvm use 16.1.2
```

Editor choice

- Visual Studio Code
- InteliJ WebStorm

Extensions

Visual Studio Code (Ctrl+Shift+X)

Angular Language Service

https://marketplace.visualstudio.com/items?itemName=Angular.ng-template

Angular Snippets - Mikael Morlund

https://marketplace.visualstudio.com/items?itemName=Mikael.Angular-BeastCode

Prettier

https://marketplace.visualstudio.com/items?itemName=esbenp.prettier-vscode

Paste JSON as Code - quicktype

https://marketplace.visualstudio.com/items?itemName=quicktype.quicktype

Optional: Prettier TypeScript Errors, Tailwind, Nx Console, ESLint, …

@angular/cli

Generator launched from the command line

https://angular.io/cli

Installation:

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

Usage:

```
ng version
```

```
ng help
```

@angular/cli toolset

Code generators

Creating a new Angular 2 project in the curent catalog

```
ng new --help

ng new yourprojectname
ng new yourprojectname --standalone --strict --routing
ng new yourprojectname --directory "."
```

Launch of a local development server

```
ng serve --help
ng serve -o
```

Generating a component skeleton (or other class ...):

```
ng generate component <name>
ng g c <name> # a shortcut
```

Code generators are so called "Schematics"

```
ng generate --help
ng g <schematic> --help

ng g <schematic>
ng g app-shell
ng g application [name]
ng g class [name]
ng g component [name]
ng g config [type]
ng g directive [name]
ng g enum [name]
ng g environments
ng g guard [name]
ng g interceptor [name]
ng g interface [name] [type]
ng g library [name]
ng g module [name]
ng g pipe [name]
ng g resolver [name]
ng g service [name]
ng g service-worker
ng g web-worker [name]
...
```

Scaffolding new components

and modules, services, etc...

```
ng generate component component-name
```

Additional flags:

- –flat - does not create a new directory for the component
- -t - Inline Template - the template places in the component file
- -s - Inline Styles - Styles places in the component file
- –spec False - omits the generation of unit tests of the component

```
ng g <schematic> --help
```

Use `ng config` or `angular.json` to set the default options

UI Toolkits

Toolsets of existing components and styles

- Angular Material Design
- Ng Zorro
- Prime NG
- Angular Bootstrap
- others

Paid "enterprise" toolkits

- Kendo
- DevExpress
- AgGrid
- others

Headless approach

- Tailwind
- Angular CDK

Bootstrap

Starting the application

```
import { platformBrowserDynamic } from "@angular/platform-browser-dynamic";
import { MyAppModule } from "./app/app.module";
```

Initialization of the main module (with the chosen chief component):

```
platformBrowserDynamic().bootstrapModule(MyAppModule);
```

The element of the main component must be in the HTML document, e.g.

```
<html>
  <body>
    <app-root> Loading... </app-root>
  </body>
</html>
```

Modules in Angular

with @NgModule

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

@NgModule({
  imports: [BrowserModule, MyModule],          // We import "Exports" from other modules

  declarations: [AppComponent, SubComponent], // Declarations for Parser HTML (directives and components)

  bootstrap: [AppComponent],                   // Component that will be the "main" application component

  exports: [],                                 // Shared elements for other modules to "import"

  providers: [],                               // Definitions of services (more on that later ...)

})
export class MyAppModule {}
```

Component declaration

```
1   import { Component } from '@angular/core';
2
3   @Component({
4     // standalone: true
5     selector: 'app-root, .extra-selector',
6     templateUrl: 'app.component.html',
7     styleUrls: ['app.component.css']
8   })
9   export class AppComponent {
10    title = 'app works!';
11  }
```

Selectors are pre-compiled with the templates:

- They have to be static
  - i.e. no `:hover, ::before` etc.
- They cannot cross element bondaries
  - no parent child - like `form > input`
- They do not re-attach at runtime

The selector is a simple CSS selector that the HTML Parser sets out on the cut elements to install this component

Declare as one of the following:

- `element-name` - by element name.
- `.class` - by class name.
- `[attribute]` - by attribute name.
- `[attribute=value]` - by attribute and value.
- `:not(sub_selector)` - only if the element does not match the sub_selector.
- `selector1, selector2` - if either selector1 or selector2 matches.

Template, Content and View

```
1   import { Component } from '@angular/core';
2
3   @Component({
4     selector: 'app-root',
5     template: `<div>
6        <h1>{{title}}</h1>
7     </div>`
8   })
9   export class AppComponent {
10    title = 'app works!';
11  }
```

"Template" places component's "view" in the parent component's view

– in the place matching by it's "selector"

View with bindings

```
1   <app-root>
2     <div>
3       <h1>app works!</h1>
4     </div>
5   </app-root>
```

Interpolates the expressions placed in the curly brackets `{{ expression }}`

Expression can only bind "local" class and template data `(no window / globals)`

No statements like:

`if, for, switch, class, function, ...` etc.

Nested components

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


@Component({
  selector: 'app-root',
  template: `<div>
    <h1>Parent</h1>
    <sub-component/>
    <sub-component/>
  </div>`
})
export class AppComponent { … }


@Component({
  selector: 'sub-componentt',
  template: `<div> <h3>Child</h3> </div>`
})
export class ChildComponent { … }
```

```
<app-root>
  <div>
    <h1>Parent!</h1>

    <sub-component>
      <h3>Child!</h3>
    </sub-component>

    <sub-component>
      <h3>Child!</h3>
    </sub-component>

  </div>
</app-root>
```

Component's templates can instantiate other components multiple times creating complex nested UI trees

So long as they are delared or imported in same module.

Routing

Angular Component Router

Routing configuration

```
1   import { RouterModule }   from '@angular/router';
2
3   @NgModule({
4     imports: [
5       RouterModule.forRoot([
6         // Default path + redirect
7         {
8           path: '',
9           redirectTo: 'todos',
10          pathMatch:'full'
11        },
12        {
13          path: 'todos',
14          component: TodosComponent,
15          children: [] // nested
16        },
17
18        // Wildcard - catch if non match
19        {
20          path: '**',
21          component: PageNotFoundComponent
22        }
23      ]),
```

The router maps the path to the URL for a fixed component.

The order of providing rules matters.

The component will appear in the template in the place indicated by the outlet:

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

Routing for Sub-Modules

```
@NgModule({
  imports: [
    RouterModule.forChild([
      {
        path: 'heroes',
        component: HeroListComponent
      },
      {
        path: 'hero/:id',
        component: HeroDetailComponent
      }
    ])
  ],
  exports: [
    RouterModule
  ]
}) class SubModuleX{}
```

RouterModule.forRoot()

- It creates a module for the main module.

RouterModule.forChild()

- It allows the submodles to have their own partial routing.

RoutingModule creates the entire module to provide the necessary tools with it

– services and directives for this routing!

Routing Lazy loading

```
const rotues: Routes = [
  { path: 'heroes',
    loadChilren: () =>
            import('./heroes/heroes.module')
            .then(m => m.HeroesRoutingModule)

    // similar to children:[] but lazy loaded
  },
]

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

Make application load faster by splitting it into lazy loaded modules

URLs with parameters

```
1   //  http://localhost/todo/15
2   { path: 'todo/:todoId', component: TodoDetailComponent }
```

Addresses can be parameterized

– by injecting ActivatedRoute or Params services, we have access to parameters

```
1   import { Router, ActivatedRoute, Params } from '@angular/router';
2   class SomeComponent{
3
4       constructor(
5           private route: ActivatedRoute,
6           private router: Router) {
7
8           const todoId = this.route.snapshot.params['todoId']
9       }
10
11      // The Router service allows for programming navigation:
12      onSelect(todo: Todo) {
13          this.router.navigate(['/todo', todo.id]);
```

Linking to the router paths

The RouterLink directive activates the path after clicking on the element.

The path can consist of many levels - given as an array:

```
<nav>
    <a routerLink="/todos" routerLinkActive="active">Todos App</a>

    <a [routerLink]="[ '/todo', todo.id ]" routerLinkActive="favourite-active">
        My favourite Todo
    </a>
</nav>
```

RouterLinkactive is a directive that adds and removes the given CSS class when the path is active or not

Binding properties and attributes directives

Property binding

```
<div [id]=" ' identificator_' + myFakeUuid " [title]="someTitle">
<div title="Hello {{userName}}">
```

Style

```
<p [style.backgroundColor]="'lime'"> I'm lime! </p>
<p [style.background-color]="'lime'"> I'm lime! </p>
```

Units

```
<p [style.font-size.px]="big? 24 : 12"> {{ big? 'big' : 'small' }}</p>
```

Classes

```
<p [class.promotion]="true"> Tu Special offer! </p>
<p class=”promotion”> Special offer! </p>

<p [ngClass]="{ promotion: false, highlight:true }"> Not here... </p>
 <p class=”highlight”> Not here.. </p>
```

Encapsulation of styles

```
1   import { Component } from '@angular/core';
2   @Component({
3     selector: 'app-root',
4     templateUrl: 'app.component.html',
5     encapsulation: ViewEncapsulation.ShadowDom,
6     styles: [
7     `h1{ color: red; }`,
8     `p{ cursor: pointer; }`]
9   })
10  export class AppComponent {
11    title = 'app works!';
```

Encapsulation 'hashes' class names to limit scope:

```
1   h1[ng_host=24tj3n9v4rhk] { color: red; }
2   p[ng_host=24tj3n9v4rhk] { cursor: pointer; }
```

Encapsulation can take the values:

- **emulated** (default) - global HTML styles are propagated to the component and its children. Syles defined in the `@Component({})` do not affect the rest of the document - **they are isolated**

- **ShadowDom**- Document styles do not affect the elements in the component or vice versa - **full isolation (see Shadow-Dom)**

- **None** - Styles work as usual, so they are propagated for the HTML document, and they are available to other components

Data transformation from PIPE

Pipe is a filter that transforms the data transferred to it:

```
<p> Total: {{ items.total | currency }} </p>
```

It can be configured by the parameters:

```
1   <p> Total: {{ items.total | currency:'PLN':true }} </p>
```

the result of one PIPE can be transferred to the next

```
1   <p> Score: {{ player.score | number | replace:'0':'-' }} </p>
```

JSON pipe

```
1   <pre>{{ SomeObiect | json }}</pre>
```

Built-in:

DatePipe, UpperCasePipe, LowerCasePipe, CurrencyPipe, PercentPipe, JsonPipe

Custom Parametrized Pipes

PIPE is a class implementing the Pipetransform interface and described by the @PiPE decorator ()

```
import { Pipe, PipeTransform} from '@angular/core';

@Pipe({
    name: 'censor'
})
export class CensorPipe implements PipeTransform {
    transform(input: string, character ? : string): string {
        return input.replace(/./g, character || '*');
    }
}
```

Can be generated with:

```
ng g pipe path/to/module/your-pipename
```

Pipes - use

To use Custom Pipe in the component, it must be declared

(just like we do with directives)

```
import { Component } from '@angular/core';
import { CensorPipe } from './censor.pipe';

@NgModule({
  declarations: [ CensorPipe, HelloComponent ]
  bootstrap: [HelloComponent]
}) class AppModule{}



@Component({
    selector: 'my-hello',
    template: '<span>{{ message | censor }}</span>',
})
export class HelloComponent {
    message: string = 'My secret sentence';
}
```

Conditions in pipes

Most pipes are stateless.

They perform transformations with pure functions, without side effects.

They work in a template like a "cache" of dynamic value.

Stateful Pipes (e.g. asyncpipe) manage the state of transmitted data

By defining a stateful pipe, we mark it as pure: false

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

It’s not recomended for performance reasons

Local references

Placing the Hash sign with the name ("#yourname") on the element creates "local references"

This variable is available **only in this component** and contains a reference to the element

(or to the component)

```
<video #movieplayer ...>...</video>


<button (click)="movieplayer.play()">
```

You can also grab child component reference :

```
@ViewChild(SomeComponentType)
child?:SomeComponentType
```

Handling events

```
class MyComponent{

  someProp = ''

  otherVariable  = ''

  someFunction(event: MouseEvent){
    this.otherVariable  = this.someProp
  }
}
```

Event use round brackets.

Event data is ALWAYS named `$event` and exists only in event handler

```
<input
  (keyup)="someProp = $any($event.target).value"/>


<button (click)="someFunction($event)">
  Click me!
</button>


Will update on click because of change detection:
{{ otherVariable }}
```

Special modifiers:

```
<input (keyup.escape)="" (keydown.enter)="" />
```

Communication with a form elements

ngModel Directive - automatically detects form input type - `input,select,textarea, ...`

Binds data with a variable:

```
1   <input [ngModel]="name">                              <---- Only update of the input values
```

Binds listening to events of change `(oninput)`:

```
1   <input ngModel (ngModelChange)=" name = $event ">     <---- only listening to input changes
```

Or for short:

```
<input [(ngModel)]="name">                              <---- Double -sided data bonding (update and listening)
```

Form in the template

The form can also be created directly in the template:

```
<form #myForm="ngForm" (ngSubmit)="save(myForm)">
  <input name="comment" [(ngModel)]="item.comment"
      required minlength="20" #commentRef="ngModel" />

    <span *ngIf="commentRef.dirty && commentRef.hasError('required')">
      Pole jest wymagane!
    </span>

    <input type="submit" value="Zapisz">
</form>
```

The form controlling the form can be obtained by using the local reference

```
<form #nazwa="ngForm" (ngSubmit)="mojaMetoda(nazwa)">
```

Own binding directives

```
@Directive({
 selector: '[myDecorations]'
})
export class MyDecorationsDirective{
    constructor(private el: ElementRef, private renderer: Renderer) { }

    // We listen to events on the host element:
    @HostListener('mouseenter') doMouseHighlight(){
       renderer.setElementStyle(
           el.nativeElement, 'backgroundColor', this.color);
    }
    // We substitute our own values to the properties of the host element:
    @HostBinding('style.textDecoration') decoration = 'underline';


    // We can use @Input and @Output too!
    @Input()
    color = 'yellow'
}
```

Custom bindings

Communication to the nested components

```
@Component({...})
export class ChildComponentNeedingData {

  // Optional Property with Type
  @Input() item?: MyTodoType;

  // Optional Property with default Value
  @Input() title = '';

  // Optional Input without the default value
  @Input() optional?: string

  // Input required by compiler (Angular 16+)
  @Input({required:true}) someProp: Type

  // Input as signal (Angular 16+)
  signalProp = input()

  signalPropRequired = input.required()

}
```

Binding in parent's component template

Passing values as a static text:

```
<todo-input title="To buy bread"></todo-input>
```

Passing **by shared reference(!)** as an expression:

```
@Component({...})
class ParentProvidingData {
    getMyItem(){
        return {
            name: 'Learn angular!',
            completed: false
        } as MyTodoType
    }
}
```

```
<todo-input [item]="getMyItem()"></todo-input>
```

Custom Event emitter

Communication to the parent component

```
@Component({
 selector: 'todo-input',
 template: '<button (click)="clickCompleted()"> ..
})
export class Hello {
 @Output() completed = new EventEmitter<boolean>()

 clickCompleted(){
  // click sends a signal to the parent
  this.completed.emit( this.todo )
 }

 // or Angular 16+
 completed = output()
}
```

Emitting events have some benefits:

- Component is reusable (can be 'connected')
- Event notifies parent to **check for changes!**

```
@Component({
 selector: 'todo-input',
 template: `
  <todo-input (completed)="saveProgress($event)"/>
 `
})
export class Hello {

  saveProgress(todo:Todo){
    // ... function performed by the subcomponent
  }

}
```

Structural directives

- ngIf - Adds and removes the template element (template) if the expression is true

```
<ng-template [ngIf]="condition">
 <div>{{ name }}</div>
</ng-template>
```

Template can be used to separate and reuse views

```
    <ng-template [ngIf]="condition"
                 [ngIfThen]="namedTemplate" />

<ng-template #namedTemplate>
    Some content
</ng-template>
```

Short version:

Putting '*' before the directive wraps element in a template for us

```
<div *ngIf="condition">
  <p> {{ name }}  </p>
</div>
```

```
<ng-container *ngIf="condition;
                     then namedTemplate
                     else otherTemplate" />

<ng-template #otherTemplate>
    Different content
</ng-template>
```

Structural directives

Allows dynamic changes to the structure of the DOM

```
<div *ngIf="completed"> ✓ </div>
```

```
<div *ngFor="let item of todoList; let i = index">
  ... itGetsRepeatedOften ...
</div>
```

```
<div [ngSwitch]="item.status">
 <p *ngSwitchCase="'completed'"> Completed <p>
 <p *ngSwitchCase="'in-progress'">
    Working on it
  <p>
 <p *ngSwitchDefault> Working on it <p>
</div>
```

Structural Directives

using templates and CommonModule directives

```
<div *ngIf="getUser() as user; else guestMessage">
  The user {{user}} is logged in
</div>

<ng-template #guestMessage>
  The user is not logged in
</ng-template>
```

```
<div *ngFor="let item of items; track: item.id">
  {{ item.name }}
</div>
```

```
<ng-container *ngSwitch="modes">
  <ng-container *ngSwitchCase="'modeA'"> ...
  <ng-container *ngSwitchCase="'modeB'"> ...
  <ng-container *ngSwitchDefault> ...
</ng-container>
```

Angular 17+ "@-Syntax"

Without ng-templates and CommonModule

```
@if ( getUser() as user;) {
  <p>The user  {{user}} is logged in</p>

} @else {
  <p> The user is not logged in</p>
}
```

```
@for (el of els; track el.id; let i = $index) {
  <div>{{i+1}}. {{ el.name }}</div>
} @empty { No items! }
```

```
@switch (condition)
{
  @case (caseA) { <p>Case A.</p> }
  @default { Default case. }
}
```

TemplateRef + ViewContainer

Allows for custom structural directives

```
@Directive({ selector: '[myShowUnless]' })
export class UnlessDirective {

 constructor(
   private templateRef: TemplateRef<any>,
   private viewContainer: ViewContainerRef
 ) { }

 @Input() set myShowUnless(condition: boolean) {
   const index = 0; // Can have multiple views!

   if (!condition) {
     // Display element from the template
     this.viewContainer.createEmbeddedView(
       this.templateRef, index, {
       // Custom binding data
       // accessed in template with `let` keyword:
       index, $implicit: {msg:'Custom data!'},
     });

   } else {
     // Destroy all created elements:
     this.viewContainer.clear();
```

Usage

```
<ng-template [myShowUnless]="someCondition"
             let-mydata
             let-i="index">
  {{i+1}}. {{mydata.msg}}
</ng-template>
```

Or shorter:

```
<ng-container *myShowUnless="someCondition;
              let mydata; let i = index">
  {{i+1}}. {{mydata.msg}}
</ng-template>
```

Content projection

Using the ngContent Directive, we can place parents HTML inside child components View

```
@Component({
 selector: 'user-profile',
 template: `
  <h4>User profile</h4>
  <ng-content></ng-content>`
})
class ContentWrapperComponent {
// ...
```

You can reference children using `@ContentChild` :

```
// ...
  @ContentChild(SomeDirective)
  childRef?: SomeDirective

  ngAfterContentInit(){
    // childRef is ready
  }
}
```

Optionally, you can define a place for an element from a specific selector:

```
<ng-content select=".my-css-selector">
  Angular 17+ allows for default content here
</ng-content>
```

```
<user-profile>
  <h5>Lito Rodriguez</h5>
  <small>Actor</small>
</user-profile>
```

```
<user-profile>
  <h4>User profile</h4>
  <h5>Lito Rodriguez</h5>
  <small>Actor</small>
</user-profile>
```

Life cycle

Directive (D) and components © can "plug into" the rendering cycle to perform their own functions:

hook:	moment wystąpienia:
ngOnChanges *(C, D)*	when a data-bound input property value changes
ngOnInit *(C, D)*	after the first `ngOnChanges`
ngDoCheck *(C, D)*	during every Angular change detection cycle
ngAfterContentInit *(C)*	after projecting content into the component
ngAfterContentChecked *(C)*	after every check of projected component content
ngAfterViewInit *(C)*	after initializing the component's views and child views
ngAfterViewChecked *(C, D)*	after every check of the component's views and child views
ngOnDestroy *(C, D)*	just before Angular destroys the directive/component

Services

and injecting dependencies in Angular

Using services

The service is every class instance that we can inject into a component or directive.

Services can:

- provide useful logic - e.g. communication with a server
- store data and share it between components
- inform components about events and provide the information you need
- and generally all other tasks not directly related to the display (view)

We do not create services ourselves.We ask Angular to provide them:

Using Injection:

Angular will find, create and provide the right object:

```
class ComponentOrService{
  constructor(

    @Inject(InjectionTokenForService)
    private someFieldName: ServiceClassName,

    // When class is also Token
    // - no need for @Inject:
    private other: OtherClass
  ){

    // ServiceClassName instance:
    this.someFieldName

    // OtherClass instance:
    this.other
  }

  // Angular 16+ function injection:
  modernService = inject(ModernToken)
```

Dependencies

By creating dependencies within our classes, we create tight coupling:

```
class ParentClass{
  // ...
  // This couples ParentClass with TodoClasss
  this.item = new Todo()
}
```

It's better to depend on abstraction and inject the concrete implementation:

```
class Todo extends AbstractMyItem {}

// It's better, but now we are now coupled to the Injector Class!
export class InjectorComponent {
  constructor(private injector: Injector) {
    const item: AbstractMyItem = this.injector.get(AbstractMyItem);
  }
```

```
// It's best when angular handles the injection:
export class InjectedComponent {
  constructor(@Inject(AbstractMyItem) private item: AbstractMyItem) { }
```

Provider configuration

We don't have to manually create `Injector` !

- Bootstrap Module itself (AppModule) creates a global Injector for us.

Ok, but how to show Angular which implementation is to be used ???

By default, it is enough to register the class so that Injector himself builds the object of this class for us along with the dependencies:

```
1  @Injectable({
2    providedIn: 'root'
3  })
4  export class OurService { }
```

The class in the Providers section will be injected wherever it was used in this module and in its children.

All consumers will share the same instance - thus making it a Singleton

Automatic injection

If our instance requires different instances the Injection will build a whole graph for us:

Just add the decorator `@Injectable` :

```
@Injectable()
export class OurService {
  constructor(@Inject(Logger) logger) {  }

  // or a class/type is enough
  // - Auto-Injecting using the type:
  constructor(private logger: Logger) {  }
}
```

Other decorators: @Component, @Directive, @Pipe, etc.

They all extend @Injectable()

Replacement of implementation

When we want to separate the implementation from the interface and inject another object, we have several options

```
@NgModule({
  // ...
  providers: [
    // Token the same as implementation ...
    AClassName,

    // Other class
    { provide: AType, useClass: AClass },

    // Ready installation, simple object,
    // or even value (e.g. config ..)
    { provide: AType, useValue: AnObject },

    // Factory function
    { provide: AType, useFactory: ( someAPI ) => {
        // ... return someAPI.someSetup()
    },

    // Dependencies for the factory function
    deps: [ SomeAPIClass ]},
  ],
})
```

Tokens

It may happen that the use of the name will be more convenient ...

We do not use strings, but `InjectionToken()`

```
const URL_TOKEN = new InjectionToken<
  ATypeToInject
>(`Descriptive name`);
```



```
@NgModule({
  providers: [
    {
      provide: URL_TOKEN,
      useValue: "http://example.com/api/v1/"
    },
    {
      provide: APIService,
      useFactory: (url) => {
        return new ApiService(url);
      }, deps: [ URL_TOKEN ]
    }
  ],
})
```

This is a unique symbol

- which, unlike the strings of characters is a reference so it avoids the risk of name collision!

Hierarchical Injector

By default, each created instance is a singleton

```
@Component({
  selector: 'isolated-data-view',
  providers: [ OurService ] // here!
})
export class AppComponent { }
```

Angular creates the object once, and then always injects the same instance.

If we want to create local instances, we define it in the providers section of `@Component` or `@Directive`

The module already registered the "ourservice" class and it can be a singleton...

However, this component now has its own Injctor, so this component and his children will receive a new Instance of the "OurService" class inaccessible to the rest of the components.

If the class is not defined here, the parent, its parent, etc. will be used.

Up to the global injector.

It its not found we will get error.

HTTP communication

@angular/common/http module

Communication with Server API:

```
1   @Component()
2   class ContactsApp implements OnInit {
3
4     contacts: Contact[] = [];
5
6     constructor(private http: HttpClient) {}
7
8     ngOnInit() {
9       // Http.get() - creates UniCast (lazy) stream
10      this.http.get('/contacts')
11
12        // Lazy queries:
13
14          // Request to the server will be completed
15          // only when subscription is made:
16        .subscribe(contacts => {
17          this.contacts = contacts
18        });
```

Do not use data sources (including HTTP) directly in components!

The components should be simple and contain only the logic for the view - a presentation layer.

Component using HTTP directly prevents reuse and exchange of data with other components

– use services for that!

Stateless, statefull and reactive

Notice that services lifecycle is usually much longer than components.

You can use that to 'cache' data and/or notify components about data changes or other important events - these are stateful and reactive services

HTTP request configuration

The HTTPClient has methods corresponding to HTTP methods: Get, Post, Put, Delete:

```
const http = inject(HttpClient);

this.http
    .post(url, payload, {
      headers: new HttpHeaders({
        // 'Content-Type': 'application/json'
      }),
      // observe: 'body',
      // responseType: 'json',
      // ...
    })
    .subscribe((data) => {
      console.log(data)
    })
```

You can extract common request features into a `HttpInterceptor`:

```
export const authInterceptor = (req, next) => {
  const auth = inject(AuthService);

  return next(
    req.clone({
      setHeaders: {
        Authorization: `Bearer ${auth.getToken()}`,
      },
    }),
  );
} satisfies HttpInterceptorFn
```

Remember to add them to Http Client Provider:

```
  providers:[
    provideHttpClient(
      withInterceptors([authInterceptor]),
      // or with HTTP_INTERCEPTOR multi token:
      // withInterceptorsFromDI()
    )
  ]
```

Reactive programming

Thanks to EventMitter and the RX.JS extension, we can work on reactive streams

RxJS

Reactive Extensions - they allow you to work on "Event streams"

Each object of type `EventEmitter<T>()` is expanding `Observable<T>()`.

The easiest way is to reason using the marble diagrams (Marbles)

Events can represent any kind of data:

- mouse clicks,
- keyboard characters,
- clock ticking or asynchronous answer from the server ...

The advantage of streams is the possibility of their repeated use

*Source: "Introduction to reactive programming you've been missing" by Andre Staltz

RxJS

RXJS has a rich library of operators enabling transformation

streams can be

- compared
- joined
- filtered
- transformed

and then subscribed to receive information when the change occurred.

The method of operation of individual operators is illustrated with "Marble Diagrams"

Examples of diagrams on : http://rxmarbles.com/

Operators can be divided into areas:

- transforming (e.g. Delay, Map, Debounce, Scan)
- connecting (e.g. Merge, Sample, StartWvith, ZIP)
- filtering (e.g. DistinctUntilchanged, Filter, Skip)
- and many others…

The operator's result is a stream

=> they can be `chained`

Subscription

It’s important to unsubscribe from unused strams

```
@Component()
class ReactiveComponent{
  api = inject(APIService)
  sub: Subscription

  ngOnInit(){
    // Subscription:
    this.sub = this.api.startRequest()
                       .subscribe()
  }

  ngOnDestroy(){
    this.sub.unsubscribe()
  }
}
```

Subscriptions can be nested:

```
const sub = new Subscription();

sub.add( this.api
             .startRequest()
             .subscribe()
       );
sub.unsubscribe() // cancel All!
```

There are also operators:

- take(n)
- takeWhile(fn)
- takeUntil(obs)
- takeUntilDestroyed() // Angular 17+
- inject(DestroyRef).onDestroy(cb)

RxJS and Angular

Angular 2 uses RXJS in several areas:

- EventEmitter, Subject
- HTTP Client module API
- changes in the value of the form are also streams

We can also create our own streams using :

- EventEmitter / Observable - to emit/broadcast your own events
- Subject - for "transferring" the stream between services and components

Easy subscription from the level of view thanks to

`AsyncPipe`

```
{{ (exampleRxStream | async)?.result }}
```

```
<div *ngFor="let item of itemsList | async">
  ...

<div *ngIf="let item; exampleRxStream | async">
  ...

<div *ngIf="exampleRxStream | async; as item">
  ...
```

```
@if( exampleRxStream | async; as item) { ... }
```

To avoid multiple subscriptions you can use:

- share()
- shareReplay()

Forms

Template Forms and Data-Driven Forms

Reactive form

```
/* We import forms module for the application module ... */
// and import it into module!
import { FormsModule }  from '@angular/forms';

import { FormControl, FormGroup, Validators } from '@angular/forms';

/* @Component ... */
export class MyForm {
  this.regForm = new FormGroup({
    username: new FormControl('Some default value', [
          Validators.required, Validators.minLength(3)])
  });
```

or we use builder:

```
class MyForm{
  constructor(private builder: FormBuilder ) {
    this.builder.group({ username: […] });

    this.regForm.value // {username: 'Johny'}
  }
}
```

Connecting with a view

FormGroup element is combined with the FormGroup directive in view

Formcontrol, on the other hand with formControlName Directive:

etc.

```
<form [formGroup]="regForm"
      (ngSubmit)="saveUser(regForm)">

      <input name="username"
             formControlName="username" />

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

By providing formControlName, we can refer to the control by its name:

```
this.regForm.get("username").value
```

Error handling

```
 @if (searchForm.get("query"); as field) {
  @if (field.pending) {
     Validating...
  }
  @if (field.hasError("required")) {
   Field is required
  }
  @if (field.getError("minlength"); as error) {
    Minimum is {{ error.requiredLength }}
  }
  @if (field.getError("censor"); as error) {
    Cannot contain {{ error.badword }}
  }
}
```

Custom methods of validation

The validator is a function that accepts the field (Control) and returning the object with the keys that are error codes and the Bolean values if the value is incorrect.

```
function startsWithLetter(control: Control): {[key: string]: any} {
    let pattern: RegExp = /^[a-zA-Z]/;

    return pattern.test(control.value) ? null : {
        'startsWithLetter': true
    };
}

this.regForm = new FormGroup({
  username: new FormControl('Some default value', [startsWithLetter])
});
```

Form states

The form and its fields can be in several states

You can check state it in HTML as CSS class or TS property on field

state:	meaning:
pristine	The field was not modified
dirty	The field was modified
touched	The field has been abandoned (blur)
valid	No validator returned the mistake
submitted	The form has been submitted
pending	The form awaits async validation

CSS classes

in the forms

Form elements also receive appropriate classes:

Class	Meaning
ng-pristine	pristine = true & dirty = false
ng-dirty	pristine = false & dirty = true
ng-touched	touched = true
ng-valid	valid = true
ng-invalid	valid = false

So we can use them in CSS:

```
input.ng-invalid.ng-dirty {
 border-bottom-color: red;
}
```

SCAM

Single Component Angular Module

```
@Component({...})
class SomeComponent{}

@NgModule({
  // Declared in same file
  delarations:[SomeComponent],
  exports:[ SomeComponent ]
  imports:[
    // Easy to import
    SomeOtherSCAMComponent
  ],
})
export class SomeComponentModule{}
```

Replaced by standalone components

```
@Component({
  standalone: true,
  imports: [
    OtherStandaloneComponent,
    ClassicModuleStillWorks,
    ...ManyComponentsList,
  ]
})
class StandaloneComponent{}
```

Component is now also a module - Simplifies imports!

Standalone

"Module-less" applications

```ts
// main.ts
import { bootstrapApplication } from "@angular/platform-browser";
import { appConfig } from "./app/app.config";
import { AppComponent } from "./app/app.component";

bootstrapApplication(AppComponent, appConfig).catch((err) =>
  console.error(err)
);
```

```ts
// app.component.ts:
@Component({
  selector: "app-root",       // Selector CSS
  standalone: true,           // Topless
  imports: [CommonModule, RouterOutlet, MyModule],
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.scss"],
})
export class AppComponent {
  title = "project";
}
```

Signals

```
const count = signal(0);

// Signals are getter functions - calling them reads their value.
console.log('The count is: ' + count());
```

To change the value of a recording signal, you can

```
count.set(3);
```

or use the .update () operation to calculate the new value from the previous one:

```
// Increment the count by 1.
count.update(value => value + 1);
```

Computed and effect

```
1  //  Signal<number>
2  const doubleCount = computed(() => count() * 2);
```

```
1  fieldName = effect(() => {
2    console.log(
3      `The current count is: ${doubleCount()}`
4    );
5  });
```

Rx Interop

```
1  fromObservable(
2    from([10,20,30]).pipe(delay(1000))
3  )
4
5  fromSignal( someSignal )
6    .pipe(...)
7    .subscribe(...)
```

Cleanup

```
effect((onCleanup) => {
  const user = currentUser();

  const timer = setTimeout(() => {
    console.log(
      `1 second ago, the user became ${user}`
    );
  }, 1000);

  onCleanup(() => {
    clearTimeout(timer);
  });
});
```

Unit tests

individual elements of the framework

Testing components

The component may require initialization

e.g. Directive from BrowserModule

```
beforeEach(() => {
  TestBed.configureTestingModule({
    declarations: [ BannerComponent ],
    imports: [ BrowserModule ],
    providers: [
      {
        provide: RealServiceToken,
        useClass: MockedService
      }
    ]
  });
  await TextBed.compileComponents()

  // creates a component and packs in a "fixture" (componentfixture)
  fixture = TestBed.createComponent(BannerComponent);
  service = TestBed.inject(RealServiceToken)

  // We can get directly to the component class:
  comp = fixture.componentInstance;
});
```

View testing

If we want to test not only the component class, but also the generated HTML, we cannot forget about the cycle of detecting changes:

```
1   it('should display original title', () => {
2
3     // Wywołaj ręcznie detekcje zmian by zakualizować widok (HTML):
4     fixture.detectChanges();
5
6     // Obiekt DebugElement posiada metody, np. do odnajdywania po CSS:
7     de = fixture.debugElement.query(By.css('h1'));
8
9     // Możemy porównać zawartość HTML z Obiektem komponentu:
10    expect(de.nativeElement.textContent).toContain(comp.title);
11  });
```

Tesing tools

- Spectator https://ngneat.github.io/spectator/
- Angular Testing Library https://testing-library.com/docs/angular-testing-library/intro/
- Jest https://ngneat.github.io/spectator/docs/jest-support

Thank you for your attention!

Questions?;-)