Angular Guidelines

Document	
Responsible	Practice Group Web Frontend Engineering
Status	Proposal
Version	1.0
Last Change	04 Nov 2020
Copyright / Licence:	

Change Control			
Version	Date	Authors	Description
1.0	04 Nov 2020	Radunz Kenneth (IT-SWE-CC1-JV4) Eterovic Teo (IT-SWE-CC1-JV7) Spirig Lukas (IT-SWE-CC2-UI) Ghilardelli Marco (IT-SWE-CC1-JV3)	Initial Version

1. Table of Contents

2. Introduction

This document provides a **non-binding foundation** for projects with a new or ongoing code base containing an Angular project.The recommendations and guidelines outlined in this document or in linked documents are not absolute and **open for discussion/changes**.

This document is an extension to https://angular.io/guide/styleguide. Instead of creating this document from scratch a well known, accepted resource is chosen as a starting point.

The following reasons led to that decision:

- decreasing maintance of this document
- adhering to well known und accepted standards

Each addition or deviation to the official styleguide is categorized by the following keywords:

- MUST: The following guideline/principle should be followed whereever possible. Exceptions are very rare
- SHOULD: The following guideline/principle should be followed if deemed applicable/beneficial.
- MUST NOT: The following guideline/principle should not be followed. Exception are very rare.

3. Content

We do not cover topics already covered in https://angular.io/guide/styleguide. The following guidelines extend or modify what is already established by the official styleguide.

3.1. Component guidelines

3.1.1. SHOULD use lifecycle hooks over non-trivial getters

A getter in angular is called once each change detection cycle. It is recommended to move calculations into lifecycle hooks.

Before

```
export class FactorialComponent {

        @Input() value: number;

        get factorial(): boolean {
                return this.calculateFactorial(value);
        }

        private calculateFactorial(n: number, accumulator: number = 1): number {
                if(n === 1) {
                        return accumulator;        return factorial(n - 1, accumulator * n);
                }

        }
}
```

After

```
export class FactorialComponent implements OnChanges {

        @Input() value: number;
        factorial = 1;

        ngOnChanges(changes: SimpleChanges): void {
                this.factorial = calculateFactorial(value);
        }

        private calculateFactorial(n: number, accumulator: number = 1): number {
                if(n === 1) {
                        return accumulator;
                }
                return factorial(n - 1, accumulator * n);
        }
}
```

3.1.2. SHOULD use pure pipes over functions in template

Do not transform data in templates using functions. This can slow down Change Detection and App performance significantly. Instead use a pure pipe if possible.

Before

```
@Component({template: '<span>{{calculateFactorial(value)}}</span>})
export class FactorialComponent {

        @Input() value: number;

        calculateFactorial(n: number, accumulator: number = 1): number {
                if(n === 1) {
                        return accumulator;
                }
                return factorial(n - 1, accumulator * n);
        }
}
```

After

```
@Component({template: '<span>{{value | factorial}}</span>'})
export class FactorialComponent implements OnChanges {

        @Input() value: number;
}

@Pipe({name: 'factorial'})
export class FactorialPipe implements PipeTransform {

        transform(value: number): number {
                return calculateFactorial(value);
        }

        private calculateFactorial(n: number, accumulator: number = 1): number {
                if(n === 1) {
                        return accumulator;
                }
                return factorial(n - 1, accumulator * n);
        }
}
```

3.1.3. MUST NOT use ViewEncapsulation.None

Global styles should not be attached to a component. Rather define them in a global css file referenced in your project's angular.json.

3.1.4. SHOULD use trackBy with ngFor

If an array is updated, the NgFor-directive updates the dom tree for all elements. This leads two uninteded side effects (e.g.: losing focus). To mitigate this issue, provide a custom trackBy function. This function should return a unique identifier for each element. Angular can now keep track of elements over multiple change detection cycles potentially reducing the amounts of dom updates.

Example:

```
<li *ngFor="let item of items; trackBy: trackByFn">{{ item }}</li>
```

```
export class MyComponent {

    trackByFn(index, item) {
        return item.id;
    }
}
```

3.1.5. MUST NOT contain business logic

A component is a represential part of your Angular application. Business logic should be moved to a service.

3.1.6. SHOULD define minimal input

Only pass information into a component that is actual needed. Convulating your component's interfaces decreases reucability.

3.1.7. MUST NOT use Injector

Do no inject the Angular Injector into your component.

Before

```
export class MyComponent {
        private readonly myService: MyService;

        constructor(injector: Injector) {
                this.myService = injector.get(MyService);
        }
}
```

After

```
export class MyComponent {

        constructor(private readonly myService: MyService) {
        }
}
```

3.1.8. MUST NOT Interact with backend

Always wrap remote resource access in a service. Calling the backend directly is a violation of SRP and decreases testability.

3.1.9. SHOULD use renderer2/viewchild for accessing dom

Do not access native dom elements directly by circumeventing Angular. Angular provides multiple mechanisms including Renderer2 and @ViewChild to interact with dom elements.

3.2. Service guidelines

3.2.1. SHOULD separate business logic / backend access

Always separate business logic and remote resources access into two separate services. This allows for an alternative implementation or mock for the remote resource access.

3.2.2. SHOULD use stateless services over top level functions

Inject business logic into your component by providing a stateless service rather than calling a top level function. This increases testability, since the service can be mocked in tests.

Before

```
function add(x: number, y: number): number {
        return x + y;
}
```

After

```
@Injectable()
export class AddService {
        add(x: number, y:number): number {
                return x + y;
        }
}
```

3.2.3. MUST NOT interact with XMLHttpRequest

Use Angular's httpClient to access remote resources as it provides additional convenience and features.

3.3. Security guidelines

3.3.1. MUST NOT trust user input

Do not pass user input into DomSanitizer's bypassSecurityTrust* methods as it can lead to XSS Vulnarabilities.

3.3.2. MUST NOT interact with DOM directly

Do not insert elements into the dom directly. Especially, if it is user input. Make use of Angular's DomSanitizer instead. Angular's bindings sanitize by default.

3.4. Rxjs guidelines

3.4.1. SHOULD always clean up subscriptions on destroy

Always clean up subscription when a component is destroyed to avoid leaks and other side effects. For further information see: Advanced subscription handling

Example

```
export class MyComponent implements OnDestroy {

        private readonly subscription = new Subscription();

        constructor(myService: MyService) {
                this.subscription.add(myService.anObservable().subscribe(val => this.doSomething(val));
        }

        ngOnDestroy(): void {
                this.subscription.unsubscribe();
        }
}
```

3.4.2. SHOULD use async pipe

The async pipe manages the subscription for you. Thus, boilerplate code is eliminated.

Before

```
@Component({
        template: '<span>{{text}}</span>'
})
export class CountComponent {

    text = '';
        private readonly subscription = new Subscription();

    constructor(textService: TextService) {
                this.subscription.add(textService.loadText().subscribe(txt => this.text = txt));
        }

        ngOnDestroy(): void {
                this.subscription.unsubscribe();
        }
}
```

After

```
@Component({
        template: '<span>{{textObservable | async}}</span>'
})
export class CountComponent {

        readonly textObservable: Observable<string>;
    constructor(textService: TextService) {
                this.textObservable = textService.loadText();
        }
}
```

3.4.3. MUST NOT nest subscribe

Subscribing to another observable in a subscribe block can lead to bugs. Additionally, it is easy to forget subscription handling for the additional subscription. Use the applicable pipeable rxjs operator instead (e.g.: mergeMap, concatMap or switchMap)

Before

```
export class TimetableComponent implements OnDestroy {

        readonly subscription: Subscription;
        readonly timetable: Timetable;

        constructor(trainService: TrainService, timetableService: TimetableService) {
                this.subscription  = trainService.trainsObservable().subscribe(train =>
                        // the timetable observables might not complete in order of subscription, which might
result in the wrong timetable shown
                        timetableService.timetableObservable(train).subscribe(timetable => this.timetable =
timetable); // unhandled subscription
                );
        }

        ngOnDestroy() {
                this.subscription.unsubscribe();
        }
}
```

After

```
export class TimetableComponent implements OnDestroy {

        readonly subscription: Subscription;
        readonly timetable: Timetable;

        constructor(trainService: TrainService, timetableService: TimetableService) {
                this.subscription  = trainService.trainsObservable()
                        .pipe(switchMap(train => timetableService.timetableObservable(train))
                        .subscribe(timetable => this.timetable = timetable);
        }

        ngOnDestroy() {
                this.subscription.unsubscribe();
        }
}
```

3.5. Styling and Layout

3.5.1. SHOULD use Scss

Scss provides many useful features and integrates well with Angular.

3.5.2. SHOULD use variables for global colors, fonts, sizes, boundaries

Reduce magic numbers by providing meaningful Scss variables.

4. Further information

This section provides link to additional resources and concepts we deem worthwile exploring. We might not agree with all the points and suggestions in these articles, but we believe they provide insight into advanced topics related to angular development.

4.1. Smart vs dumb components

https://medium.com/@jtomaszewski/how-to-write-good-composable-and-pure-components-in-angular-2-1756945c0f5b

4.2. State management

https://medium.com/@2muchcoffee/angular-state-management-a-must-have-for-large-scale-angular-apps-8b98e5a761c7 https://redux.js.org/understanding/thinking-in-redux/three-principles

4.3. Change detection

https://blog.angular-university.io/how-does-angular-2-change-detection-really-work/

https://netbasal.com/a-comprehensive-guide-to-angular-onpush-change-detection-strategy-5bac493074a4

https://www.mokkapps.de/blog/the-last-guide-for-angular-change-detection-you-will-ever-need/

4.4. Lazy loading

https://angular.io/guide/lazy-loading-ngmodules

4.5. Communication between components

https://angular.io/guide/component-interaction

4.6. ng-template, ng-container and ngTemplateOutlet

https://blog.angular-university.io/angular-ng-template-ng-container-ngtemplateoutlet/

4.7. Component lifecycle

https://angular.io/guide/lifecycle-hooks

4.8. Running outside angular zone.js

https://medium.com/@krzysztof.grzybek89/how-runoutsideangular-might-reduce-change-detection-calls-in-your-app-6b4dab6e374d

4.9. Module organization

https://levelup.gitconnected.com/where-shall-i-put-that-core-vs-shared-module-in-angular-5fdad16fcecc

4.10. Service scopes

Global/Core Provider, Module/Lazy Loading Provider, Component/View Provider

https://codecraft.tv/courses/angular/dependency-injection-and-providers/ngmodule-providers-vs-component-providers-vs-component-viewproviders/

4.11. Advanced subscription handling

https://medium.com/angular-in-depth/the-best-way-to-unsubscribe-rxjs-observable-in-the-angular-applications-d8f9aa42f6a0

4.12. Security

https://netbasal.com/angular-2-security-the-domsanitizer-service-2202c83bd90