Angular Coding-Guideline

Der Guideline beschreibt den verwendeten Coding-Stil von Angular als eigene Richtline.

---

1. Integration des Window-Objektes

Begründung:
Es soll keine Referenz auf das globale Objekt erfolgen.

Aktion:

```
import { Inject } from "@angular/core";
import { DOCUMENT } from "@angular/common";

export class MyClass {
  private window: Window;

  constructor(@Inject(DOCUMENT) private document: Document) {
    this.window = this.document.defaultView;
  }

  foo() {
    console.log(this.document);
    console.log(this.window);
  }
}
```

---

2. Einheitliche Code-Formatierung und Linting

Begründung: Teamübergreifende Code-Ausgabe
Aktion: Prettier in Intellij-Einstellung aktivieren, dann Konfiguration teilen. TSLint und Lint mit Husky werden für das Linting verwendet.

---

3. Imports-Sorting

Begründung: Bessere Lesbarkeit der Imports
Aktion:
An den Styleguide von Angular halten (https://angular.io/guide/styleguide#import-line-spacing):
Laut dem offiziellen Angular-Styleguide sollten Importzeilen alphabetisch angeordnet werden, destrukturierte Importsymbole sollten ebenfalls alphabetisch sortiert werden. Eine Leerzeile trennt Importe von Drittanbietern und die "eigenen" Anwendungsimporte voneinander.

---

4. Subscribe To Memory-Leak

Begründung: Während einer Subscription reißt zum Beispiel die Internet-Verbindung ab, es kann zu Memory-Leaks kommen, wenn dieses nicht behandelt wird.

Es gibt dafür unterschiedliche Implementierungen, am besten erscheint eine automatisierte Lösung mit einem eigenen Decorator.

Der Decorator:

```
function AutoUnsub() {
    return function(constructor) {
        const orig = constructor.prototype.ngOnDestroy
        constructor.prototype.ngOnDestroy = function() {
            for(const prop in this) {
                const property = this[prop]
                if(typeof property.subscribe === "function") {
                    property.unsubscribe()
                }
            }
            orig.apply()
        }
    }
}
```

Implementierung des Decorators:

```
@Component({
    ...
})
@AutoUnsub
export class AppComponent implements OnInit {
    observable$
    ngOnInit () {
        this.observable$ = Rx.Observable.interval(1000);
        this.observable$.subscribe(x => console.log(x))
    }
}
```

---

5. Keine Business-Logik in Komponenten

Begründung: Angular ist ein MVC-Framework, die Komponenten dienen nach dem MVVM-Patten lediglich der Entgegennahme von Ereignissen und dem Setzen von HTML-Properties.

Lösung:
In den Komponenten werden lediglich Properties verwendet und Ereignisse behandelt. Geschäfts-Logik wird in Services verarbeitet.

---

6. Keine Code-Kommentare und Todos

Begründung:
Der Code sollte für sich selber in seiner Einfachheit lesbar sein. Todos werden für temporäre Kommentierungen verwendet.

---

7. State-Management und RxJS

Die Anwendung enthält zu viel an Boilerplate-Code, wenn nicht eine vollständige NgRx-Implementierung von Anfang an erfolgt. Es wird Elf mit seinen Zusatz-Bibliotheken als STORE-Tool mit RxJs bei Bedarf verwendet. Der Code enthält keine Async-Calls oder Promises.
